2000

Reengineering of Legacy Systems to Distributed Environments.

Miguel Angel Serrano-vargas
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

REENGINEERING OF LEGACY SYSTEMS TO DISTRIBUTED ENVIRONMENTS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Miguel A. Serrano
B.S., Universidad Autónoma Metropolitana, 1992
M.S., Louisiana State University, 1998
M.S. in Sy.Sc., Louisiana State University, 1999
May 2000

To my family

ACKNOWLEDGMENTS

During my journey through the Ph.D. program, many people and organizations made it possible for me to finish this work. I take this opportunity to thank all of them, although words are not enough to express my deep appreciation towards them.

First, I would like to thank my parents. They are among the wisest teachers I have. Throughout my life, they have taught me to be honest, truthful, appreciative and hardworking; to do my best in any endeavor I choose to undertake; to appreciate life; and to help others. Their unconditional love and encouragement were the greatest source of strength. I also thank my brothers and sisters, my aunt Lourdes, the Mondragon family, and my other relatives; without them this work would have not been possible. For their love and help, I am thankful.

I extend my sincere appreciation towards my major professor, Dr. Carver for her support and good advice. She has been a source of constant inspiration and motivation during this time. Her dedication, professionalism and humanity have evoked my deepest respect and admiration. I also would like to thank the other members of my committee, Dr. Jones, Dr. Kraft, Dr. Tang, and Dr. Sundar, thank you all for your time and your good advice.

Special thanks to the assistantship committee in the Computer Science department at Louisiana State University for appointing me as a graduate assistant. Thanks to the International Service Office in LSU for the Fulbright scholarship. To the Institute of International Education in México, thanks for granting me with the Fulbright scholarship, and special thanks to Conacyt (National Council for Science and Technology of México) for the financial support during these years.

Last but not least, I want to thank all my friends in LSU who made my stay more pleasant; to my friends from the Mexican Student Association, the Society of Hispanic Professional Engineers, to Pilar, Alberto, Miriam, Blanca, Alicia, and Pan; thank you all for your friendship and love. Thanks to my brothers Luis, Tomas and Francisco for the very special e-mail network that we created. It was always joyful to read all your messages. My special gratitude to Elaine, for her friendship, love, encouragement and support. Finally, special thanks to Carlos Montes De Oca for his great help and his priceless advice during these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The object-oriented paradigm and client/server and distributed technologies have become widely used in the last decade. There is an increasing interest to migrate and reengineer legacy systems to these new hardware technologies and software development paradigms. Software engineers who wish to reengineer such legacy systems face challenges, such as lack of documentation and programs that are difficult to comprehend. Middleware technologies such as CORBA and DCOM make the development of new distributed systems, as well as the migration of legacy systems to distributed platforms, more feasible. Distribution of a system consists of two parts: 1) subsystem decomposition and 2) allocation of the subsystems to different sites.

In this research, we define a reengineering environment that assists with the migration of legacy systems to distributed environments. We define a reengineering methodology that uses reverse engineering, software metrics, clustering, and data mining to migrate legacy systems to object-based distributed environments. The reengineering environment includes the methodology and an integrated set of tools that support the implementation of the methodology. The methodology consists of multiple phases. First, we use reverse engineering techniques for program comprehension and design recovery. We then decompose the system into a hierarchy of subsystems by defining relationships between the entities of the underlying paradigm of the legacy system. The decomposition is driven by data mining, software metrics, and clustering techniques. Next, if the underlying paradigm of the legacy system is not object-based, we perform object-based adaptations on the subsystems. We then create components by wrapping objects and defining an interface. Finally, we

allocate components to different sites by specifying the requirements of the system and characteristics of the network as an integer-programming model that minimizes the remote communication. We use middleware technologies for the implementation of the distributed object system.

CHAPTER 1. INTRODUCTION

In the last decade, there has been increasing use of the object-oriented paradigm, distributed systems, and client/server technologies, suggesting that distributed object systems will represent a significant portion of the next generation of software systems. As the object-oriented paradigm and distribute systems become more widely used, there is also increasing interest to integrate, migrate, and reengineer legacy systems to these technologies. Existing legacy systems use a variety of software paradigms (e.g., unstructured: COBOL, structured: C, and even object-oriented: C++) and a variety of hardware platforms. The motivation to migrate legacy systems to new technologies and paradigms (e.g., distributed objects) is more than the obvious advantage of being able to use and share remote computational resources. In some legacy systems, reengineering of the legacy software to increase maintainability and to increase the potential to use modern tools and techniques represents another key motivation.

The object-oriented paradigm offers advantages such as modularity, extensibility, and reusability. From the distributed programming point of view, objects are good candidates for modeling units of distribution because they encapsulate attributes and methods to act as independent entities communicating through passing messages. These characteristics resemble the features of a distributed system [Guer99].

Industry and academia are adopting the combination of the object-oriented paradigm and distributed technology for the development of new systems. Object-oriented and distributed systems are having a major impact on areas such as databases [Bert90] and programming languages [Wals98]. They are also impacting scientific computing [Ishi97], [Luck97] that in the past was confined to the use of imperative

languages such as FORTRAN and C, parallel programming constructs for parallel computers, or message passing interfaces such as PVM and MPI [Geis94] for scientific distributed computing.

The reengineering and development of an application for distributed systems can be viewed as a two step procedure: 1) subsystem decomposition of the system into suitable distributable units (i.e., fragmentation) and 2) placement of the subsystem on processing units (i.e., allocation) [Ceri83], [Pura98b]. Recent middleware technologies such as Common Object Request Broker Architecture (CORBA) [OMG 99a] and Distributed Component Object Model (DCOM) [Micr98] make the implementation of distributed systems more feasible.

The research objective of this work is to show that we can produce a methodology together with a set of tools to assist in the incremental migration of legacy systems to distributed environments. Consequently, our hypothesis is: Migration of legacy systems to distributed environments can be achieved in a systematic way with the support of a reengineering environment consisting of a methodology and a set of tools.

In this research we combine reverse engineering, software metrics, dependence analysis, data mining, and clustering techniques. We develop a semiautomatic evolutionary reengineering and migration methodology that maps a legacy system to an object-based distributed system. First, we use reverse engineering techniques for the architectural and design recovery. Then, we use data mining and clustering techniques to produce a hierarchical data cohesive decomposition of the system to appropriate distributable units. The relationships in the underlying paradigm of the legacy system drive the data mining and clustering algorithms. We use object-oriented

metrics if the underlying paradigm is object-oriented, and program-uses-file relationships otherwise. We then perform wrapping and object-based adaptations to the subsystems to address encapsulation and information hiding if the underlying paradigm of the legacy system is not object-based. We allocate components to different sites by specifying the requirements of the system and characteristics of the network as an integer-programming model that minimizes the remote communication. IDL (Interface Definition Language) is used to specify the interface of the components and CORBA is used to implement the communication among the distributed subsystems. The approach that we present is semiautomatic in the sense that the methodology can not be fully automated, and besides, knowledge of a human expert is necessary during the migration and reengineering process.

This thesis is organized as follows. Chapter 2 provides the related work and background information. Chapter 3 presents the migration approach. Chapter 4 presents the reengineering environment that supports the migration approach. Chapter 5 presents case studies to demonstrate the feasibility of the approach, and Chapter 6 presents the summary and conclusions of this research.

CHAPTER 2. BACKGROUND AND RELATED RESEARCH

In the last two decades, there has been increasing interest in research on the migration and reengineering of legacy systems to newer technologies and paradigms (e.g., reengineering of unstructured code to the structured approach, and migration of the structured approach to the object-oriented paradigm). Distributed objects is a relatively new technology and there have been few proposals of methodologies to migrate legacy systems to distributed objects environments in a systematic way, instead, several ad-hoc guidelines to perform the migration have been proposed.

The methodology for the migration of legacy systems to distributed object environments that we propose in this research consists of several phases (e.g., reverse engineering, subsystems decomposition, and allocation). Object-oriented metrics, data mining, and clustering help drive the decomposition of legacy systems into subsystems with low inter-subsystem communication.

In this chapter, we present background information as well as a review of the literature of related techniques and approaches to this research. Section 2.1 presents definitions of some of the terms that we use in our work. Sections 2.2, 2.3, 2.4, and 2.5 present related research and background in the areas of migration of legacy systems, object-oriented software metrics, data mining, and clustering respectively. Section 2.6 presents an overview of related work to this research.

2.1. Definitions and Concepts

In the literature related to the reverse engineering, reengineering, forward engineering, and software engineering fields, there are many conflicting uses in the terminology. To provide consistency, the Taxonomy Project of the IEEE-CS

Technical Council on Software Engineering (TCSE) - Committee on Reverse Engineering has been working on a unified taxonomy of the field [TCSE97], [Chik90]. In a similar effort, The Joint Logistic Commanders Computer Resources Management group (JLC/CRM) authorized and sponsored a Department of Defense (DoD) policy workshop in Santa Barbara, California in 1992. Now known as SB-1, this workshop formally defined software reengineering terminology for the DoD [JLC 92].

In this section, we list the definitions of some of the terms that we use in this research.

- Forward engineering: The traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system [Chik90]. In [JLC 92] it is defined as the set of engineering activities that consume the products and artifacts derived from legacy software and new requirements to produce a new target system.
- Reverse engineering: The process of analyzing a subject system with two goals in mind: (1) to identify the system's components and their interrelationships; and, (2) to create representations of the system in another form or at a higher level of abstraction [Chik90].
- Reengineering: Examination of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [Chik90]. In [JLC 92] it is defined as the examination and alteration of an existing subject system to reconstitute it in a new form. This process encompasses a combination of sub-

processes such as reverse engineering, restructuring, redocumentation, forward engineering, and retargeting.

- Restructuring: Transformation from one form of representation to another at the same relative level of abstraction. The new representation is meant to preserve the semantics and external behavior of the original representation [Chik90].
- Redocumentation: Form of restructuring where the resulting semantically equivalent representation is an alternate view intended for a human audience [Chik90].
- Design recovery: Subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system. The objective of design recovery is to identify meaningful higher-level abstractions beyond those obtained directly by examining the system itself [Chik90].
- Retargeting: Process of transforming and hosting or porting the existing system in a new configuration. The new configuration could be a new hardware platform, a new operating system, or a CASE platform [JLC 92].
- Software evolution: Process of adapting an existing software system to conform to an enhanced set of requirements [Ruga99].
- Refactoring: Behavior-preserving manipulations that change the design of the code to make it more reusable. Refactorings are typically design-level changes that facilitate the reuse of the software without altering the behavior.

2.2. Migration of Legacy Systems

Legacy systems are systems that utilize languages, platforms, and techniques that do not represent current technology. Most organizations have legacy systems that serve critical business needs. Legacy systems have problems such as high maintenance cost and missing or out-of-date documentation. The challenge is to keep the legacy application running while converting it to newer, more efficient code that makes use of new technology and programming paradigms.

Legacy systems can include not only old software systems running in old computers, but also newer systems written in modern programming paradigms. In reference to object-oriented systems, Bar [Bar99] states: "The law of software entropy dictates that even when a system starts off in a well-designed state, requirements evolve and customers demand new functionality, frequently in ways the original design did not anticipate. A complete redesign may not be practical, and a system is bound to gradually lose its original clean structure and deform into a bowl of object-oriented spaghetti". This loss of structure together with the lack of up-to-date documentation and other factors makes the migration and reengineering of legacy systems necessary.

Currently, many organizations are migrating and reengineering their legacy systems to new programming technologies and paradigms. There are several approaches to migrate legacy systems to new platforms [Snee98]. One approach is the total redevelopment of the legacy system starting with new specifications. There are several advantages (specifications, design, and implementation can be started with good practices) and disadvantages (high cost, time consuming, high risk, and re-

starting years of knowledge and business rules existing in the legacy system and not documented anywhere). Cimitile considers that it is infeasible that large legacy systems can be redeveloped from scratch and achieved by "one-shot" replacement [Cimi98]. Evolutionary migration and reengineering is a more feasible approach. Evolutionary migration applies reverse engineering to the existing code to recover the architectural and design models. Two approaches can then be followed: a new software system can be developed or the components of the legacy system can be reused [DeLu97].

2.2.1 Reengineering of non Object-Oriented Legacy Systems

Programs written for first-generation computers were written in low-level first-generation languages, such as assembly language. Later, some of those programs had to be migrated to higher-level programming languages such as FORTRAN. The code written in the early FORTRAN and COBOL compilers lacked modularity. Millions of lines of code of spaghetti code were written, making the code difficult to understand and maintain. Consequently, a new wave of reengineering and migration of many of those programs took place to migrate non-structured and spaghetti code to modular languages. In the last decade, we have witnessed a new reengineering and migration wave to change from the modular-structured paradigm to the object-oriented paradigm. Now, the latest trend is to migrate to framework and component-based distributed systems. These shifts of paradigms over the years have produced a significant amount of research in the area of software engineering.

In [Burd96], [Burd98], a number of techniques such as reverse engineering and data dependences are used to identify reusable modules at different levels of

granularity from legacy COBOL systems. They evaluate approaches to predict the number of potential reuse candidates that are available within a legacy application. Sneed and Majnar [Snee96a], [Snee98] report experiences on migrating legacy systems to object technologies and client/server systems. They use wrapping at different levels of encapsulation such as job, transaction, program, module, and procedure. De Lucia [DeLu97] presents an approach to migrate legacy systems to object-oriented platforms. They use reverse-engineering techniques for object identification. They then encapsulate the identified objects into wrappers. Finally, they perform the incremental translation of "legacy-objects" to object-oriented platforms. Lakhotia [Lakh99] presents techniques for restructuring functions with low cohesion into functions with high cohesion. They restructure functions by breaking them into smaller cohesive pieces by partitioning the set of output variables on the basis of their pairwise cohesion. Then, they use program slicing to create slices that replace the original function. Luksh deploys software engineering techniques for reengineering and restructuring parallel scientific applications by migrating FORTRAN systems to High Performance FORTRAN (HPF) and C++ on a network of workstations and massive parallel processors [Luck97].

2.2.2 Reengineering of Object-Oriented Legacy Systems

Although the object-oriented paradigm is relatively new, there are many object-oriented legacy systems. Factors such as the lack of experience, improper training, lack of tool support, and extensive maintenance activities have made these object-oriented systems become classified as legacy systems. In the past, most of the reengineering efforts were focused on traditional paradigms (e.g.,

transforming/reengineering spaghetti code to modular code, and non object-oriented code to object-oriented code). Now, with the existence of millions of lines of code in object-oriented legacy systems, the reengineering of object-oriented legacy systems is an active area of research. Early adopters of the object-oriented programming paradigm now face the problem of transforming their object-oriented legacy systems into the more reusable and flexible frameworks and components. The ability to deal with large and poorly documented object-oriented programs definitely requires support from tools as well as methodologies [Deme99b].

There have been several efforts to produce tools and methodologies for the reengineering and migration of object-oriented systems. The Renaissance consortium [Renn98], supported by the European Union Information Technology program (ESPRIT) [Espr95], and partners from the industry and academia, has the goal of improving the applications management, evolution, reengineering and reuse processes. Renaissance is distinct from other reengineering projects by its focus on architectural evolution and the recovery of designs of system families in 4GLs rather than the more common COBOL or FORTRAN.

Another effort towards the reengineering of legacy systems and object-oriented legacy systems is proposed in [Dewa99], [Stev98]. They argue that "the main problem in not that the necessary expertise does not exist, but rather, that is hard for engineers to become experts in all necessary areas". They propose the use of reengineering patters (RP) to help to codify and disseminate expertise.

In software design, the term pattern has been imported from architecture [Alex77] to describe an application of an expert solution to a common problem in context. A

design pattern [Gamm94] describes a solution for a recurring design problem in a form that facilitates the reuse of a proven design solution. An important element of a design pattern is its discussion of the advantages and disadvantages of applying the pattern.

A RP is a description of an expert solution to a common system reengineering problem, including its name, context, and advantages and disadvantages. A RP embodies expertise about how to guide a reengineering project to a successful conclusion. It connects an observable problem in the code to a reengineering goal and describes the process of going from the existing problem legacy solution to a refactored solution which meets the reengineering goal. A RP gives a method appropriate for a specific problem, rather that a general methodology [Duca99].

In the FAMOOS [Famo99] project, Ducasse [Duca99] uses reengineering patterns to move from the legacy systems solutions to new refactored solutions. They propose reengineering patterns for type-check elimination, changing architectural dependencies, and transforming inheritance into composition.

There have been other approaches to reengineer and restructure object-oriented systems that show potential design problems (e.g., large deep of inheritance, multiple-inheritance, high coupling, and low cohesion). Alkadi and Carver [Alka98] propose a set of techniques driven by object-oriented metrics to evaluate object-oriented designs. They use object-oriented metrics to identify potential design problems and to help restructure the design to conform to predetermined design criteria. Demeyer [Deme99b] proposes a hybrid approach combining visualization methods with object-oriented metrics to help drive the reverse engineering of a system and to identify potential design anomalies. Sassen [Sass99] explains the use of a metrics-based

reengineering tool (Audit-Reengineering) for model capture and problem detection of object-oriented legacy systems. They use a set of five object-oriented metrics to assess the suitability of the metrics from the perspective of reengineering. Harrison [Harr98c] presents a set of object-oriented metrics collectible from design documents that facilitate the easy extraction of the design model and assist in making subsequent reengineering decisions.

2.3. Object-Oriented Metrics

Object-oriented metrics are an integral part of object-oriented technology. Object-oriented metrics are used for purposes such as cost and effort estimation, productivity, and quality of software (e.g., design quality, maintainability). For the purpose of this research, object oriented metrics are used to determine relationships and measure dependencies between object-oriented constructs. We use object-oriented metrics to drive a decomposition of the subject system into subsystems with low coupling and therefore low communication when the subsystems are allocated to different processing sites/nodes in a distributed environment.

Metrics in the object-oriented paradigm is an active field of research. Features such as polymorphism and inheritance that make the object-oriented paradigm powerful also make it more complex than the imperative and structured paradigm. The set of software metrics defined for the imperative and structured paradigm such as Lines of Code (LOC) and McCabe's cyclomatic complexity [Fent97] are not sufficient for measurement of object-oriented systems. As a result, new metrics and measurement frameworks have been defined for the object-oriented paradigm [Basi96], [Chae98], [Chid94], [Harr98a], [Hend96], [Bria98b], [Bria99a].

Chidamber and Kemerer provided the early work in object-oriented metrics [Chid94], [Chid91]. They proposed a suite of six metrics for object-oriented design based on measurement theory. They offer some empirical validation based on data collected from a field study. Table 1 shows the metrics proposed and the items measured [Hend96]. Basili presents the result of a study that empirically investigates the suite of object-oriented metrics introduced by Chidamber and Kemerer [Chid94] and evaluates the metrics as predictors of fault-prone classes and quality indicators [Basi96]. Welch defines a set of object-based metrics at different levels of granularity (application, module, operation, and statement). Metrics such as coupling, cohesion, object-orientedness, maintainability, and potential concurrency among objects are used to define a set of techniques for identifying potential concurrency among and within objects [Welc96b].

Table 1. Metrics proposed by Chidamber and Kemerer

Metric	Description	Item Being Measured
WMC	Weighted methods per class	Size and complexity
DIT	Depth of inheritance tree	Size
NOC	Number of children	Size/coupling/cohesion
CBO	Coupling between objects	Coupling
RFC	Response for a class	Communication and complexity
LCOM	Lack of cohesion in methods	Internal cohesion

One problem with some of the frameworks and metrics suites is that they use different notations and formalisms; thereby, it is difficult to compare them. In

addition, some of the frameworks have some ambiguity in their definitions. To remedy the situation, Briand provides a unified framework for coupling and cohesion metrics in object-oriented systems. He reviews existing frameworks and provides a standardized terminology and formalism for expressing new and existing measures in a fully consistent and operational manner [Bria98b], [Bria99a].

Following is a set of definitions of some of the metrics proposed by the frameworks and suites.

- CBO was first defined in [Chid91] as: "CBO for a class is a count of the number of non-inheritance related couples with other classes". An object of a class is coupled to another if methods of one class use methods or attributes of the other. A refined definition of CBO proposed in [Chid94] is: "CBO for a class is a count of the number of classes to which it is coupled ... this includes coupling due to inheritance".
- RFC is the set of methods that can potentially be executed in response to a message received by an object of that class.
- MPC is the number of send statements defined in a class. The number of send statements sent out from a class may indicate how dependent the implementation of the local method is on the methods in other classes.
- DAC is the number of Abstract Data Types (ADTs) defined in a class. The number of variables/attributes having an ADT type may indicate the number of data structures dependent on the definition of other classes.
- NOC is the number of direct descendants for each class

- LCOM is the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. The metric is set to zero when the subtraction is negative.
- DIT is the maximum depth of the inheritance graph of each class.

2.4. Data Mining

Data mining is the process of extracting valid, previously unknown, comprehensible, and actionable information from large databases; and using it to make crucial business decisions [Simu96]. Fayyad [Fayy96a] defines the term KDD (Knowledge discovery in Databases) as the overall process of extracting high-level knowledge from low-level data or the non trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data. Some sources use the terms data mining and KDD interchangeably. There are multiple names to refer to KDD including information harvesting, data archeology, functional dependency analysis, knowledge extraction, and data pattern analysis [Fayy96a].

In this research, we use data mining to drive the decomposition of legacy systems into data cohesive subsystems. We mine relationships between software constructs present in the legacy code (e.g., program uses file). The idea is to mine relationships to look for association rules. An association rule is an implication of the form "when a program uses file X then the program also uses file Y *c%* of cases and this pattern is present in *s%* of transactions", *c* is the confidence and *s* is the support of the mining rule. Confidence measures the fraction of times that Y exists in the data when X is present (statisticians refer to this as the conditional probability of Y given X). Support

measures the number of transactions that contain X and Y as a fraction of the total number of transactions.

The results of the mining phase allow us to produce groups of programs that use a similar set of data files, thus, grouping data cohesive components. The data cohesive subsystem decomposition produces distributed systems with high, intra-subsystem communication and low, inter-subsystem communication. Details of how data mining is used in our research are given in Section 3.2.1.

Data mining encompasses a broad set of technologies, including data warehousing, database management, data analysis algorithms, and visualization [Apte97]. Data mining and KDD are not new areas. They evolved from the intersection of research in the fields of databases, machine learning, pattern recognition, statistics, artificial intelligence and reasoning with uncertainty, knowledge acquisition for expert systems, data visualization, information retrieval, and high performance computing [Fayy96b].

Data mining and similar techniques are required as we go deeper into the age of digital information. Our ability to analyze and understand massive datasets lags far behind our ability to gather and store data. For many years, computer systems have been accumulating data, current databases are huge, and they are still growing rapidly. For example, the US retailer Wal-Mart handles more than 20 million transactions per day and the NASA's Earth Observing System will produce several gigabytes per hour by the end of the century [Fayy96a]. The process of digging into these huge volumes of data to identify significant knowledge is not a trivial task. This task is especially difficult if we do not know that particular information or knowledge exists or can be

deduced from the database. Thus, it is evident that intelligent automated tools are needed to analyze, profit, and extract useful information from these large databases.

The KDD process involves numerous activities and steps. Data mining is one part of the overall process of knowledge discovery in databases. Fayyad [Fayy96b] includes the following steps in the KDD process:

(1) Understand the application domain. This step involves collecting all the relevant knowledge about the domain and defining the objectives of the analysis.

(2) Create a target data set. Determine the set of variables, partition of the data, or sample on which the discovery process will be done.

(3) Perform data cleaning and preprocessing. Elimination of noise and outliers, as well as the definition of strategies to handle missing values, noise, time sequence information, and normalization.

(4) Perform data reduction and projection. Find the appropriate form to represent the data or variables. Use transformations to reduce the number of variables, or to project them to spaces where the discovery process has a greater possibility for success.

(5) Choose the data mining task. Decide the type of knowledge (patterns) to be mined. Classifications, regressions, clustering, sequences, and associations are examples of the patterns that can be mined.

(6) Select the data mining algorithms. Choose the method (algorithm) to extract the patterns selected in the previous step.

(7) Perform data mining. Search for the desired patterns using the selected algorithms. This step involves selecting the parameters to run the algorithms.

(8) Interpret the mined results and iterate over steps 1 to 7 to improve the mined results.

(9) Consolidate the results into discovered knowledge and resolve possible conflicts between known knowledge and the mined knowledge.

This particular sequence of steps does not imply a sequential process but rather an iterative one. Indeed, the KDD process is open to multiple loops (iterations) among these steps.

Data mining algorithms can be divided in three major categories based on their nature to extract information: predictive modeling (also called supervised learning or classification), clustering (also called unsupervised learning or segmentation), and frequent pattern extraction [Apte97]. Supervised learning models use a portion of the data set for training. Two very important issues in data mining are the model functions and the model representation. Some of the more common models functions in data mining are [Fayy96b], [Mont99]:

- Classification. Maps (or classifies) a data item into one of several predefined categorical classes.
- Clustering. The idea is to group data items to form classes or clusters of data items according to some similarity function or probability density models. In this case, the data mining algorithm defines the classes as opposed to classification where the classes are predefined. For instance, data mining clustering algorithms

can be used to identify groups of homogeneous people to help develop a marketing plan.

- Regression. The objective is to map a data item to a prediction variable to predict the value of a certain set of attributes.
- Summarization. The aim is to find a compact description of the data set. An example of this function is the derivation of summary rules.
- Dependency modeling. The objective is to find significant dependencies among data items. The dependency can be expressed in structural terms or in quantitative terms. The former describes a dependency network, and the latter describes the strength of the dependency using a particular scale.
- Link analysis (also called frequent pattern extraction). The objective is to find relationships among the data items. For example, link analysis may produce association rules. The objective is to find rules of the form "c% of the customers that buy product A also buy products B and D." This kind of information can be used to design the floor plan of the store, the marketing strategy, or even to forecast inventory levels.
- Sequence analysis. The objective is to model patterns that occur over time. The idea is to model the required states that produce a particular sequence of events. One example of sequence analysis is the identification of sequences of events such as "if event *A* occurs, then *c%* of the time events *B* and *D* occur within the next *t* units of time." This information can be used to forecast equipment failures and stock booms.

- Change and deviation detection. The objective is to detect significant changes in data from a previously time-stamped state of the data.

There are several techniques and models that can be used to represent these data mining functions, including decision trees and rules, neural networks, nearest-neighbor classification algorithms, case-based reasoning, and Bayesian networks [Fayy96b].

2.5. Clustering

Clustering is the grouping of data items to form classes or clusters of data items according to some similarity function or probability density models. The algorithm used for clustering defines the classes as opposed to classification where the classes are predefined. Clustering explores the inherent tendency of a point pattern to form sets of points (clusters) in the multidimensional space [Bajc98]. A cluster is a group of *objects* whose members are more homogeneous to each other than to the members of any other group [Ouya96a]. The term object could be anything that could be represented as a point in a multidimensional measurement space.

In this research, clustering analysis techniques are used in the formation of hierarchies of subsystems during subsystem decomposition. The subsystem decomposition of non object-oriented legacy systems uses a clustering algorithm to form hierarchical sets of programs and files. The clustering algorithms are guided by the mined associations, and the internal connectivity of the sets is controlled by similarity functions.

There have been several approaches and methodologies proposed to decompose legacy systems into subsystems with higher cohesion and lower coupling. Clustering

of a software system can be viewed as a graph partition problem. In a good partition, highly interdependent modules (nodes) are grouped in the same subsystem (clusters) and, independent modules are assigned to separate subsystems. Mancoridis states this clustering problem as an optimization problem and solves it using hill-climbing and genetic algorithms [Manc99], [Soud99]. Clustering techniques were used by [Ouya96b], [Ouya96a] to present a method for enhancing reusability in the design phase. They specify the system with Z schemas. Then, clustering techniques are applied to cluster the schemas that have potential for a high degree of similarity at the design stage. Anquetil [Anqu99] uses clustering techniques to achieve software remodularization. They present an excellent work for the application of different clustering algorithms and similarity metrics specially suited for software systems.

There are four important issues to consider for the clustering of entities or objects, (1) object description (attributes), (2) object relationships (associations), (3) similarity metrics, and (4) clustering algorithms [Anqu99]. We elaborate on these issues in the following paragraphs.

1. Object description can be based on (1) formal descriptive attributes (the attribute has direct impact on the software system behavior) such as functions, variables, and types and (2) non-formal descriptive attributes such as the naming of variables and comments.
2. Object relationships can be classified as direct and sibling. An object has a direct relationship with another entity when they depend on each other (e.g., class A inherits from class B). Two objects have a sibling relationship when they have the

'same behavior', or they have the same kind of relationship to a third entity (e.g. if class A and B inherit from class C, then, they have a sibling relationship).

3. Similarity metrics measure the resemblance of two objects. Similarity metrics can be classified in four categories, (1) association coefficients which compares the references two objects have in common only considering whether dimensions are zero/absent, (2) distance coefficients where objects are treated as points and the coefficient computes the distance (e.g., Euclidean distance), (3) correlation coefficients which are based on linear correlations between values for all dimensions, and (4) probabilistic coefficients which are based on the probability that two objects are similar given their respective vectors.

   In this research, we focus on similarity metrics based on association coefficients. Figure 1 shows three examples of association coefficients.

Let X and Y be two objects and F the set of all possible dimensions

$a = |X \cap Y|$

$b = |X \setminus Y|$

$c = |Y \setminus X|$

$d = |F \setminus (X \cup Y)|$

Jaccard: $sim(X,Y) = a / (a+b+c)$

Simple matching: $sim(X,Y) = (a+d) / (a+b+c+d)$

Sorence-Dice: $sim(X,Y) = 2a/(2a+b+c)$

Figure 1. Association coefficients similarity metric

4. Clustering algorithms are classified in two major categories: partitional and hierarchical [Guja98]. Partitional clustering algorithms try to determine k partitions that optimize a certain criterion function. The square error criterion is the

most commonly used. The square error is a good measure of the within-cluster variation across all the partitions. A hierarchical clustering is a sequence of partitions in which each partition is nested into the next partition in the sequence. An 'agglomerative' algorithm for hierarchical clustering starts with the disjoint set of clusters, placing each input data point in an individual cluster. Pairs of items or clusters are then successively merged until the number of clusters reduces to *k*. At each step, the pairs of clusters merged are the ones between which the distance is the minimum. Clustering algorithms are differentiated by the way they compute the distance of a new cluster to all the other ones. There are three main distance functions used commonly: d_{mean} (centroid-based approach), d_{max} (max. all-points approach, complete linkage, or furthest neighbor), and d_{min} (min. all-points approach, single linkage, or closest neighbor). With d_{mean} as the distance measure; the pair of clusters whose centroids or means are the closest are merged at each step. With d_{max}, the pairs of clusters merged are the ones containing the furthest pair of points. With d_{min}, the pairs of clusters merged are the ones containing the closest pair of points.

2.6. Distributed Object Systems

Distributed objects systems (DOS) emerge from the intersection of the object-oriented paradigm with client/server and distributed systems technologies. A computer system is considered 'distributed' when the programs and data that programs work on are spread out over more than one computer, usually over a network. In DOS, the units of distribution are objects that contain methods (behavior) and attributes (state).

In recent years, with the increasing popularity of the Internet and economical driving forces such as e-comerce, there has been an increasing interest to develop efficient distributed systems. DOS is a promising technology to use for these systems.

Since DOS is a relatively new technology, the research community and the industry are trying to set standards for this new approach. There are different candidates programming languages (e.g., C++, Java, Ada, and SmallTalk), distributed object platforms (e.g., CORBA [OMG 99a], DCOM [Micr98], and RMI [Sun 99]) and economic interests in the industry (e.g., Microsoft/DCOM vs Sun-Netscape-IBM/Corba) to consider [Kava98]. Most of the distributed object platforms available now offer similar features. In this research we do not address issues regarding the distributed object platform or the object-oriented programming language. We focus on other issues such as the methodology for the migration and integration of the legacy systems with the systems developed with the new technology, and the efficient allocation of objects/components to different sites in the network.

Bastarrica proposes an architectural specification to serve as the basis for obtaining optimal distribution of object-oriented application components over a target network that minimizes remote communication between components. They propose a Binary Integer Programming (BIP) model with constraints such as storage and communication to provide optimal distribution [Bast98].

Purao proposes an approach for systematically deriving distributable units from an object-oriented system and effectively distributing them to processors. They use logical specifications of the object-oriented system, usage patterns, and semantic associations to derive an efficient decomposition and allocation [Pura98b].

2.7. Related Work Overview

This chapter presented background and a research survey in the areas of reengineering of legacy systems, object-oriented metrics, data mining, clustering, and distributed object systems.

A summary of the different approaches presented in this chapter is given in Table 2. It contains several columns evaluating the different research projects or methodologies that we presented in this chapter. The first column contains the features that we evaluated. The features are divided in two categories; (1) goals: shows the goal of the research or methodology (e.g., migration, restructuring, and integration), and (2) techniques: shows the techniques that were used to fulfill the goals (e.g., dependences, wrapping, clustering, and software metrics).

Table 2. Summary of related work

	[Burd96] [Burd98]	[Snee96a] [Snee96b] [Snee98]	[DeLu97]	[Lakh99]	[Alka99]	[Bar99] [Famo99] [Deme99a] [Deme99b] [Sass99]	[Mont99] [Mont98a]	[Pura98a] [Pura98b]	[Bast98]	[Manc99] [Soud99]
Goal										
Legacy Systems to OO	X		X							
OO to Dist. Systems (DS)								X	X	
Legacy Systems to Dist. Sys.		X								X
Restructuring non OO	X	X		X			X			
Restructuring OO					X	X				
Subsystem decomposition	X		X				X			
Integration (of systems)		X								X
Optimal allocation in DS								X	X	
Security DS										X
Techniques										
Code dependences	X		X				X			
Wrapping		X	X					X	X	X
Slicing			X	X						
OO Metrics			X		X	X				
Clustering							X			X
Data Mining							X			
Optimization								X	X	
Integrated tool environment	X	X	X	X	X	X	X	X		X

CHAPTER 3. THE MIGRATION APPROACH

In this research, we define a methodology for the evolutionary migration of legacy systems to object-based distributed environments. The approach, visualized in Figure 2, is summarized in the following steps:

(1) Apply reverse engineering and design recovery techniques to obtain the software architecture of the underlying legacy code.

(2) Decompose the system into suitable units for distribution driven by use/dependence relationships. Software metrics, data mining, and dependence analysis techniques are used in this step. We use clustering techniques to represent the system as a hierarchy of subsystems.

(3) For non object-oriented legacy systems, perform the object-based adaptation, addressing encapsulation and information hiding.

(4) Define the wrapping and interface definition of each of the subsystems. The new units (subsystem + wrapper + interface) are called components. IDL is used in this approach.

(5) Allocate components to processing units or sites in the network by minimizing inter-component communication.

(6) Implement the distributed system using middleware technologies for distributed systems such as CORBA that allow the 'transparent' communication between components.

Notice that the order of steps 5 and 6 can be changed, since the allocation (reallocation) of the components can be done after the implementation of middleware technology.

This approach is suitable for evolutionary migration because once the components have been defined (step 5-6), reengineering can be applied to each of them independently. We describe each of the five steps of the methodology in Sections 3.1 - 3.6.

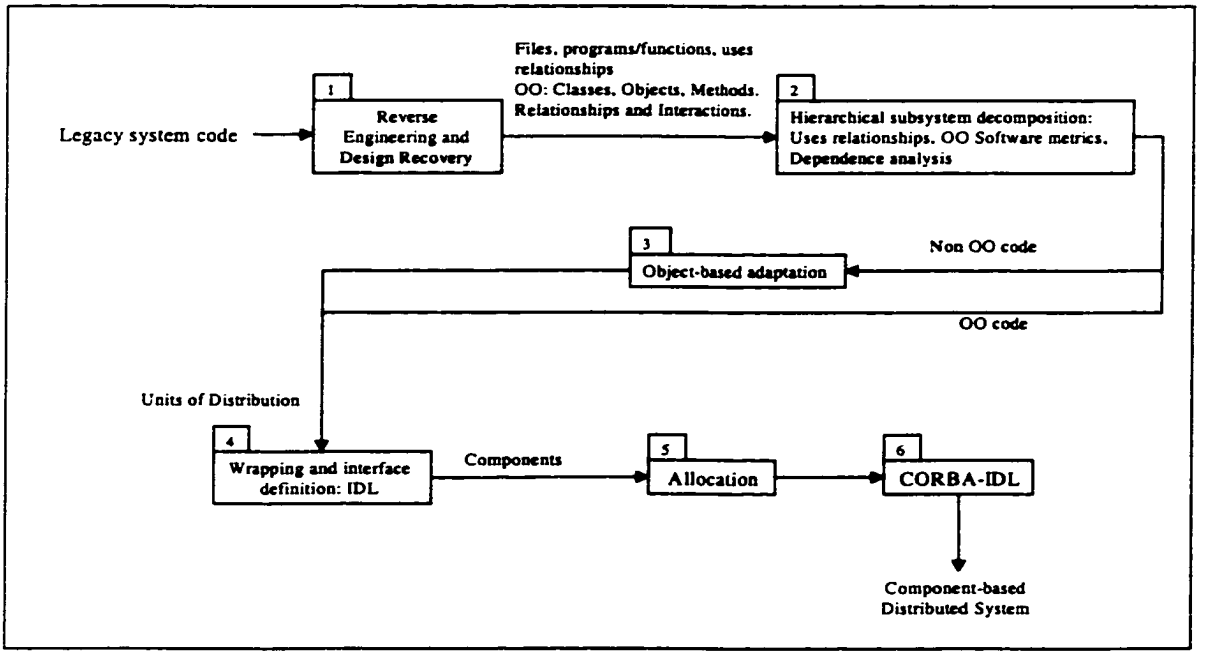

Figure 2. Migration methodology

3.1. Software Architecture and Design Recovery of the System

To recover the software architectural model, the code of the original system must be parsed and techniques for the analysis of the code must be applied. For non object-oriented legacy systems, we are interested in identifying portions of code (programs, functions, methods) called programs, and independent data repositories called files. We are also interested in identifying uses and call relationships between such entities (e.g., program uses file, program calls program).

For object-oriented legacy systems, we identify the basic object-oriented constructs (e.g., classes, objects, and methods), their physical dependences and their logical dependences. Object-oriented systems have complex relationships and interactions among entities (object-oriented constructs). Relationships and interactions between the different entities such as class-attribute, class-method, and method-method contribute to dependences and structural complexity of the system.

3.2. Subsystem Decomposition

Identification of subsystems in the legacy code is the goal of this step. Since the subsystems will be deployed in a distributed environment, the ultimate goal of the decomposition approach is to generate subsystems that minimize communication.

One of the main concerns in any distributed system is the inter-site communication between processes in different nodes. Therefore, we are especially interested in relationships present in the system that produce communication and dependence between entities (files, programs, objects, methods, and classes).

The objective of this step is to generate a hierarchical, data-cohesive subsystem decomposition of the original legacy system. The techniques for the decomposition of non object-based and object-based legacy systems are different [Mont99], [Serr00]. We explain these decomposition techniques in Sections 3.2.1 and 3.2.2.

3.2.1 Subsystem Decomposition of non Object-Based Legacy Systems

In this section, the objective is to generate a subsystem decomposition of a non object-oriented legacy system. We use data mining and clustering techniques to drive the reverse engineering and decomposition of the subject non object-oriented legacy system into a hierarchy of data cohesive subsystems. We use mining functions from

the IBM intelligent miner [IBM 98] and the ISA methodology, developed by Montes de Oca [Mont99], [Mont98a].

The IBM intelligent miner (IM) is a suite of statistical, processing, and mining functions for analyzing data in large databases. The processing functions in the IM include calculation, discretization, filtering, and joining. Mining functions include associations, classification, clustering, sequential patterns, time sequences, and prediction. Statistical functions include bivariate analysis, factor analysis, linear regression, and component analysis. In this research, we focus mainly in the use of association rules mining functions.

ISA is a system-level methodology that decomposes a software system into a hierarchy of data cohesive subsystems. ISA uses data mining techniques to guide the subsystem formation process. The ISA methodology identifies subsystems in three steps [Mont99]:

1. Build a database view of the system. A database view of the system is a representation of the system or a subset of it using a database.
2. Perform data mining. Use a data mining algorithm to mine association rules over the data base view of the system.
3. Consolidate and interpret results. Combine the outcome of the mining process and use clustering techniques to produce a subsystem decomposition of the target system.

The generic system that ISA can decompose into subsystems is a system composed of multiple identifiable portions of code, called *programs*, and multiple independent data repositories, called *files*. That is, ISA accepts a software system S

composed of a set of programs $\mathcal{P}$ and a set of data files $\mathcal{F}$. A typical example of such a system is a human resources system written in COBOL. This system would likely be composed of several subsystems such as payroll, training, recruiting, and benefits. Each subsystem would include several programs and several files. For instance, the payroll subsystem may include programs to print the payroll, to print checks, to perform the calculations, and to report tax withheld. In addition, the payroll subsystem would contain the roster file, the salaries file, and the scheduling file. Moreover, each subsystem may be decomposed into several sub-subsystems. Finally, the whole system may also include files that are used by several subsystems such as the master employee file and the organizational units file.

The outcome of ISA is the decomposition of $\mathcal{S}$ into data cohesive subsystems. A subsystem is defined as a set of programs and files. In other words, a subsystem is a set $Z=\{G, H\}$ such that $G \subset \mathcal{P}$ and $H \subset \mathcal{F}$. ISA decomposes $\mathcal{S}$ into k subsystems $Z_i=\{G_i, H_i\}$ for $i= 1,2, \ldots, k$, where $G_i \cap G_j = \varnothing$, and $H_i \cap H_j = \varnothing$ for $i, j = 1, 2, \ldots, k$, and $i \neq j$. However, $G_1 \cup G_2 \ldots \cup G_k$ may not be equal to $\mathcal{P}$, and $H_1 \cup H_2 \ldots \cup H_k$ may not be equal to $\mathcal{F}$ because there are some programs that cannot be classified into any subsystem and some files that are used by several subsystem (i.e. the master employee file in the example above). The subsystems that ISA generates have the characteristic that for each Z_i, the programs in G_i access primarily the files in H_i. In other words, the programs in a subsystem use predominantly the files in the subsystem. However, this subsystem decomposition does not imply that a program in a subsystem Z_i cannot use a file in a subsystem Z_j. Rather, it means that the files in a

subsystem are used predominantly by the programs in the same subsystem. That is, the programs in a subsystem access the same data repositories. In that sense, ISA produces data cohesive subsystems. In addition, ISA organizes the k subsystems into one or more hierarchies of subsystems. ISA joins the identified subsystems to form larger subsystems (i.e., suprasystems containing subsystems). These larger subsystems are merged to form even larger subsystems. This process continues until the largest subsystems reach a dissimilarity threshold. This threshold is based on a similarity function. An example of such a function is the following. Let f_1 be the set of files that are accessed by programs in subsystem S_1, and f_2 the set of files accessed by programs in subsystem S_2. Then, S_1 and S_2 are merged if $|f_1 - f_2| < c$, where c is a user-defined parameter that defines the dissimilarity threshold.

ISA forms the hierarchy using two types of subsystems: *primitive subsystems* and *complex subsystems*. Primitive subsystems contain programs and files, and complex subsystems contain primitive subsystems and files. Primitive subsystems correspond to leaf nodes, and complex subsystems correspond to internal nodes in the trees that represent the hierarchical subsystem decomposition.

There is a distinction between the files assigned to primitive subsystems and the files assigned to complex subsystems. A file assigned to a primitive subsystem Z_i is used primarily by the programs in Z_i. A file assigned to a complex subsystem S_i is used primarily by programs in primitive subsystems that have S_i as an ancestor. Empirically, a complex subsystem is a set of primitive subsystems. Therefore, files assigned to a complex subsystem can be seen as shared files in the sense that programs in different primitive subsystems uses these files.

ISA decomposes the system into one or more subsystem hierarchies. The resulting decomposition may not be a single hierarchy tree but a forest. In this case, there are several complex subsystems with no parent subsystem (i.e., several roots). These subsystems are called *main subsystems*. Figure 3 shows an example of the type of decomposition that ISA produces.

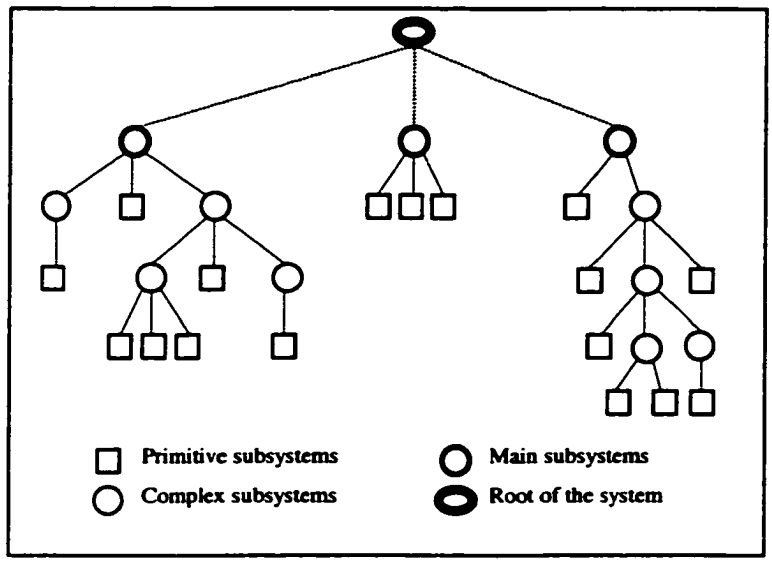

Figure 3. Example of an ISA decomposition

ISA produces other interesting outcomes and byproducts such as unconnected programs, singular programs, unconnected files, common files, independent files, hierarchies of file implications, and link files. Unconnected programs and singular programs are programs that were not assigned to any particular subsystem. The difference between them is that unconnected programs are not considered in the data mining process. Unconnected programs do not enter in the mining process because they use less than two files, thereby they do not produce any association with other program. Singular programs are programs that entered the mining phase but produced no association with any other program.

Similarly, unconnected files are files that are used by less that two programs thereby they are not included in the mining process and consequently they are not assigned to any subsystem. Some files cannot be assigned to any particular subsystem because they are used by programs from different subsystems. These files are called *common files*. An example of a common file is the master employee file in a human resource system. The master employee file cannot be assigned to any particular subsystem (e.g., payroll, benefits, scheduling) because this file is used by all the subsystems.

Finally, some of the files assigned to a particular subsystem may be used by programs in other subsystems. These files are called *link files*. A link file is assigned to a particular subsystem X because most of the programs that use it are in X. Nevertheless, this link file is used by few programs outside X. Table 3 summarizes the information produced by ISA.

One advantage of the ISA methodology is that it is automatic. The decomposition of the subject system can be done automatically. Therefore, ISA is capable of processing large software systems. ISA only needs the source code to produce the subsystem decomposition.

Now we explain the process by which ISA produces the subsystem decomposition. The process consists of three major phases: I. Create a database view of the system. ISA uses a set of tuples as the database view of the system, II. Perform data mining over the database view of the system. ISA mines the set of tuples in search for association rules, and III. Consolidate the results of the data mining process into a high-level abstraction. In this phase, ISA uses four algorithms to produce the

outcome described in the previous section. Details about the ISA methodology can be found in [Mont99].

Table 3. ISA output description

Product	Description
Primitive subsystem	Subsystem composed of a set of programs and n files, where $n \geq 0$
Complex subsystem	Subsystem composed of at least one primitive subsystem, n complex subsystems, and m files, where $n \geq 0$ and $m \geq 0$
Main subsystem	Complex subsystem that is not an element of any other complex subsystem (i.e., complex subsystem with no parent)
Unconnected programs	Programs that are not included in the mining process
Singular programs	Programs included in the mining process but not assigned to any primitive subsystem
Unconnected files	Files that are not included in the mining process
Common files	Files included in the mining process but not assigned to any subsystem
Link files	Files assigned to a particular subsystem but used by programs outside that subsystem

We describe how these three main phases work.

I. Create a database view of the system. The database view of the system is a matrix or table representation where program-uses-file relationships present in the system are exposed. To generate the database view of the system, first we assign unique identifiers to the files and programs revealed by the previous design recovery step, and then we generate a matrix representation of the system.

- Assign identifiers. In this step, the programs and files in the system are assigned unique identifiers to facilitate their manipulation. Additionally, for each program, a list containing all the files that the program uses is produced.

- Generate the matrix representation of the system. The matrix representation of the system consists of two sets of tuples, called the alpha sets (alphaT set and alphaN set). The alphaT and alphaN sets can be viewed as matrices or tables. AlphaT is a matrix in which each row represents a file and each column a program. AlphaN is the transpose of the matrix representing AlphaT. The matrices contain the value of one when the program represented by a row/column uses the file represented by the column/row and a value of zero otherwise. The programs and the files in the matrix representing the alpha set is a subset of the programs and files identified during the parsing. The alpha set contains only programs and files that are capable of forming associations during the mining process. Therefore, files not used by any program, files used by only one program, programs not using any file, and program using only one file are not included in the alpha set.

II. Perform data mining. In this phase, 2-dimensinal associations are mined from the alpha sets. A 2-dimensional association has the form s[p, q] where s is the support of the association, and p and q are either programs (if alphaT is used) or files (if alphaN is used). In this phase, we mine the alphaN set and the alphaT set. Mining the alphaN set produces associations that relate two programs using a common set of files. For example, an association that results from mining the alphaT set may be 15[34 78]. This association means that programs 34 and 78 use 15 files in common. Mining the alphaT set produces associations that relate two files. Here, the interpretation is different. The associations are used to create association rules. For example, if a mined

association is 10[17, 41] and the confidence of the association rule 41 → 17 is large, then this information is used to create a hierarchy of file implications. In this particular case, the implication means that if a program uses file 41 it also uses file 17.

III. Consolidate and interpret results. In this phase, the two sets of associations produced in phase II are used to guide the clustering process to produce the hierarchical subsystem decomposition. The process to generate this hierarchical subsystem decomposition is as follows:

- Form groups of programs and create hierarchies. Use the mined associations from the alphaT set to guide a clustering process that forms groups of programs. The associations guide a merging process in which the groups are joined to form larger groups. The result of this process is a series of trees. Each tree is a hierarchy of groups. Leaf nodes in the trees represent primitive subsystem, non-leaf nodes represent complex subsystems, and root nodes represent main subsystems. However, the nodes in the trees (i.e., groups of programs) are not yet subsystems because they do not contain files.
- Assign files to these groups. Assign each file *f* in the alpha set to the group of programs (i.e., a node in the forest) that contains more programs using *f*. After applying the assign-files algorithm, the groups contain programs and files. At this step, the subject system has been decomposed into a hierarchy of data cohesive subsystems.
- Form hierarchies of file implications using common files. Some of the files in the alpha set cannot be assigned to any particular group of programs because

these files are used in many groups (i.e., common files). These common files may form hierarchies of file implications. The associations mined from the alphaN set are used to identify hierarchies of file implications among the common files.

- Merge main subsystems to create a single tree of subsystems. At this point, the subject system has been decomposed into several main subsystems. Each of these main subsystems is decomposed into a hierarchy of subsystems. We now need a single hierarchical tree to maintain consistency and to provide more information on how the main subsystems are related. The process works by merging main subsystems to form larger subsystems (i.e., supra-subsystems). This merging process continues until a single supra system is produced. The criterion used to create the hierarchy is file coverage. A main subsystem *B* is considered to be a child of main subsystem *A* if most of the files used by programs in *B* are used by programs in *A*.
- Represent the subsystem decomposition. Show the subsystem decomposition in textual or graphical form.

3.2.2 Subsystem Decomposition of Object-Based Legacy Systems

The objective is to generate a subsystem decomposition of an object-based legacy system. We achieve this by following a procedure similar to the one used for the subsystem decomposition of non object-based legacy systems. The most important difference is that the interactions and relationships that occur in object-based systems are not as simple as the programs-uses-file interaction used to drive the subsystem decomposition of non object-based legacy systems. Instead, we consider class-class,

class-method, and method-method relationships. We say that two object-oriented constructs are related if they share some relationship or interaction. The steps that we follow for the subsystem decomposition of object-based legacy systems are:

(1) Generate sets of related object-based constructs such as objects, classes, and methods.

(2) Use object-oriented metric techniques to generate metric sets.

(3) Define interaction matrices from the metric sets.

(4) Use data mining algorithms (association rules) over interaction matrices

(5) Use hierarchical clustering algorithms over association coefficients to produce a hierarchical decomposition of the target system.

Object-oriented systems have complex relationships and interactions among the object-oriented constructs (i.e., classes, objects, and methods). Interactions such as class-attribute, class-method, and method-method contribute to dependences and structural complexity of the system. Coupling measures the strength of the association among modules/classes. Many mechanisms contribute to the coupling of classes such as message passing (method invocation), inheritance, and polymorphism. Cohesion is a metric that measures the degree to which elements of a module belong together. Low coupling and high cohesion are two desired features in any well-design system.

We are interested in decomposing an object-oriented system into subsystems that are suitable for distribution [Serr99b]. There are different levels of granularity for the units of distribution, such as large grain (subsystems, applications), medium grain (classes, objects, and methods), and fine grain (single class attributes, line of code) [Pura98b]. We use the class as a unit of distribution and we preserve inheritance for

inter-site distribution. One of the main concerns in any distributed system is the inter-site communication between processes in different nodes. In object-oriented systems, we are interested in discovering relationships present in the system that produce communication and dependence between classes, methods, and attributes. Classes that show low coupling also show low message-passing. Object-oriented metrics such as Coupling Between Objects (CBO), Response for a Class (RFC), and Data Abstraction Coupling (DAC) provide this kind of information.

Briand [Bria99a] defines formalisms for expressing object-oriented software metrics. In this research, we use and extend his notation and formalisms. Figure 4 contains these terminology and formalisms. We are interested in CBO, RFC, and DAC metrics:

- CBO is defined as the count of the number a classes to which a class *c* is coupled. A class is coupled to another if methods in one class use methods or attributes in another class. We use four versions of CBO: CBO(c) and CBO'(c), taken from [Bria99a], and CBO_d(c) and CBO_d'(c), which we define. These metrics are formally defined in Figure 5.
- RFC is defined as follows: "RFC = |RS| where RS is the response set of a class... The response for a class is a set of methods that can potentially be executed in response to a message received by an object of that class" [Chid91].
- Data Abstraction Coupling (DAC) is defined in [Li 93] as "the number of ADTs defined in a class". ADT is a class in that context. The number of variables (attributes) having an ADT type may indicate the number of data structures dependent on the definitions of other classes. We use the following definitions of

DAC: DAC(c) and DAC'(c), taken from [Bria99a], and DAC_t(c), which we define. The definitions are given in Figure 5. The definitions of the metrics are based on the terminology and formalisms in Figure 4.

In our definitions, CBO, DAC, and RFC are defined as the size of a set (e.g., CBOSet and DACSet). We call these sets 'metric sets'.

The subsystem decomposition process consists of four steps. We now explain these steps:

1. Generate sets of related object-based constructs such as objects, classes, and methods. From the design recovery phase, we create a list of pairs of related object-oriented constructs (objects, classes, and methods). The design recovery phase gives us information of directly related pairs of entities with relationships such as class-class, class-method, and method-method.
2. Use object-oriented metric techniques to generate metric sets. In Figure 5, the object-oriented metrics (CBO, RFC, and DAC) are defined as the size of the respective metric sets (CBOSet, RFCSet, and DACSet). In this step, we are not interested in the size of the metric set, but rather in the elements of the metric set. The CBO and DAC metric sets give us information about the interaction between one class and all the other classes in the system. In the case of the RFC set, we have the interaction between one class and the methods of different classes. The importance of using object-oriented metrics as opposed as just using information from the design recovery is that object-oriented metrics take into account features such as inheritance, polymorphism, and transitive relationships.

C: Set of classes in the object-oriented system

- For each class $c \in C$

Parents(c), *Children*(c), *Ancestors*(c), *Descendents*(c): Set of parents, children, ancestors, and descendants classes of class c respectively.

$M(c)$: The sets of methods of class c

$M_D(c) \subseteq M(c)$: Set of methods declared in c, i.e. methods that c inherits but does not override or virtual methods of c

$M_I(c) \subseteq M(c)$: Set of methods implemented in c, i.e. methods that c inherits but overrides or nonvirtual noninherited methods of c.

$M(c) = M_D(c) \cup M_I(c)$ and $M_D(c) \cap M_I(c) = \varnothing$

$M_{INH}(c) \subseteq M(c)$: Set of inherited methods of c

$M_{OVR}(c) \subseteq M(c)$: Set of overriding methods of c

$M_{NEW}(c) \subseteq M(c)$: Set of noninherited, nonoverriding methods of c

M(C): Set of all methods in the system: $\cup_{c \in C} M(c)$

- For each method $m \in M(C)$

Par(m): Set of parameters of method m

- Let $c \in C$, $m \in M_I(C)$ and $m' \in M(C)$

$SIM(m)$: Set of statically invoked methods of m.

$m' \in SIM(m) \Leftrightarrow \exists\, d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' is invoked for an object of static type class d.

$NSI(m, m')$: Number of static invocations of m' by m

Let $m' \in SIM(m)$. $NSI(m, m')$ is the number of method invocations in m where m' is invoked for an object of static type class d and $m' \in M(d)$

PIM(m) : Set of polymorphically invoked methods of m.

$m' \in PIM(m) \Leftrightarrow \exists\, d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' may, because of polymorphism and dynamic binding be invoked for an object of dynamic type d.

$NPI(m, m')$: Number of polymorphic invocations of m' by m

Let $m' \in PIM(m)$. $NPI(m, m')$ is the number of method invocations in m where m' can be invoked for an object of dynamic type class d and $m' \in M(d)$

- For each class $c \in C$

$A(c)$: Set of attributes of class c

$A(c) = A_D(c) \cup A_I(c)$ where

$A_D(c)$: Set of attributes declared in class c (i.e., inherited attributes)

$A_I(c)$: Set of attributes implmented in class c (i.e., noninherited attributes)

$A(C)$: Set of all attributes in the system: $\cup_{c \in C} A(c)$

- For each method $m \in M(C)$

$AR(m)$: Set of attributes referenced by method m

BT: Set of built in types provided by the programming language (e.g., integer, real)

UDT: Set of user-defined types (e.g., records, enumerations, but not classes)

T: Set of available types in the system: $T = BT \cup UDT \cup C$

$T(a)$: Type of attribute a where $a \in A(C)$. $T(a) \in T$

$Par(m)$: Set of parameters for method m where $m \in M(C)$

- For each parameter $v \in Par(m)$

$T(v)$: Type of parameter v. $T(v) \in T$

$Lv(m)$; Set of local variables of method m where $m \in M(C)$

- For each local variable parameter $w \in Lv(m)$

$T(w)$: Type of local variable w. $T(w) \in T$

Figure 4. Terminology and formalisms

3. Define interaction matrices from the metric sets. Now we generate matrices of interacting classes. For interacting classes, the rows and the columns contain the classes in the system. The intersection of row *i* and column *j* have the value of 1 if class *i* and class *j* belong to the metric sets (i.e., if the two classes have some interaction). The intersection has the value of 0 otherwise.
4. Use data mining algorithms (association rules) over interaction matrices. Perform association rules data mining algorithms using the interaction matrices as the input for the data mining and obtain the support and confidence of the association rules.
5. Use hierarchical clustering algorithms over the association coefficients (support and confidence from the data mining) to produce a hierarchical decomposition of the system. From Section 2.5 we note that there are four important issues to consider when clustering objects, (1) object description, (2) object relationships, (3) similarity measures, and (4) clustering algorithms. The objects we are dealing with are classes. The interaction matrices specify the object description and relationships. The rows represent the objects (classes) and the columns the description. The underlying metric (e.g., CBO, DAC) specifies the relationship. The object description is based on formal descriptive attributes; specifically, an object (class) is described by the classes with which it has a relationship. We use sibling object relationships (e.g., class c1 inherits from class c2). We use the support of the associations rules as the similarity measure (association coefficients). Finally, we use hierarchical clustering to generate the hierarchical subsystem decomposition. Hierarchical clustering generates sequence of partitions (subsystems) in which each partition is nested into the next partition. A subsystem

in this context is a set of one or more classes. The hierarchical clustering begins with disjoint set of clusters and places each class in an individual cluster. Pairs of clusters are then successively merged until the number of clusters reduces to k, where k is the number of clusters desired. At each step, the pairs of clusters merged are those between which the distance is the minimum. Therefore, at each step we merge the clusters that have the group of most related or 'similar' sets of classes. We use d_{min} (min all-points based approach) as the distance function, i.e., the pairs of clusters merged are the ones containing the closest pair of classes.

The decomposition technique is a bottom-up approach. We start by creating single subsystems consisting of single classes. We then create subsystems that contain related entities (classes). The resulting problem of 'composing' subsystems is less complex than the problem of decomposing the original entire system (top-down).

3.3. Object-Based Adaptation of Subsystems

ISA generates a set of subsystems organized hierarchically and a set of entities that do not fit any specific subsystem (i.e., common files, unconnected files, unconnected programs, and singular programs). These subsystems and entities are still under the same paradigm as the legacy system. Therefore, object-based adaptations are done to assure that object-oriented principles such as encapsulation and information hiding are followed.

The object-based adaptation produces an object-based representation of the system consisting of object-subsystems and object-entities. In this work, an object is a set of methods, files, and an interface that contains the definition (i.e., the prototype) of the services (public methods) that the object provides.

-Let $c \in C, d \in C$

CBO(*c*): (coupling between objects) Count of the number of classes to which class *c* is coupled. A class is coupled to another if methods in one class use methods or attributes of the other

$CBO(c) = |CBOSet(c)|$

$CBOSet(c)$: $\{ d \in C - \{c\} \mid uses(c, d) \vee uses(d, c) \}$

uses(*c*, *d*): A class *c* uses a class *d* if a method implemented in class *c* references a method or attribute implemented in class *d*. It can be defined for dynamic method invocations as in a)and for static method invocations as in b)

a) $uses(c, d) \Leftrightarrow (\exists m \in M_I(c): \exists m' \in M_I(d) : m' \in PIM(m)) \vee (\exists m \in M_I(c): \exists a \in A_I(d) : a \in AR(m))$

b) $Uses\_s(c, d) \Leftrightarrow (\exists m \in M_I(c): \exists m' \in M_I(d) : m' \in SIM(m)) \vee (\exists m \in M_I(c): \exists a \in A_I(d) : a \in AR(m))$

CBO'(*c*): Count of the number of classes to which class *c* is coupled without counting coupling due to inheritance

$CBO'(c) = |CBOSet'(c)|$

$CBOSet'(c)$: $\{ d \in C - (\{c\} \cup Ancestors(c)) \mid uses(c, d) \vee uses(d, c) \}$

CBO_d(*c*): Count of the number of classes to which class *c* is di-coupled. A class *c* is di-coupled to class *d* if methods in *c* use methods or attributes of *d*

$CBO\_d(c) = |CBOSet(c)|$

$CBOSet\_d(c)$: $\{ d \in C - \{c\} \mid uses(c, d) \}$

CBO_d'(*c*): Count of the number of classes to which class *c* is di-coupled without counting di-coupling due to inheritance

$CBO\_d'(c) = |CBOSet\_d'(c)|$

$CBOSet\_d'(c)$: $\{ d \in C - (\{c\} \cup Ancestors(c)) \mid uses(c, d) \}$

- *Let* $R_0(c) = M(c)$ *and*

$R_{i+1}(c) = \cup_{m \in R_i(c)} PIM(m)$

RFC(c): (response for a class) Number of methods that can potentially be executed in response to a message received by an object of class c

$RFC_d(c) = /RFCSet_d(c)/$

$RFCSet_d(c) = \cup_{i=0 \text{ to } \alpha} R_i(c)$

DAC(*c*): (data abstraction coupling) Number of noninherited attributes in class *c* having a class as their type

$DAC(c) = |DACSet(c)|$

$DACSet(c) = \{a \mid a \in A_I(c) \wedge T(a) \in C\}$

DAC'(*c*): Number of classes used as types for noninherited attributes in class *c*

$DAC'(c) = |DACSet'(c)|$

$DACSet'(c) = \{ T(a) \mid a \in A_I(c) \wedge T(a) \in C \}$

DAC_t(*c*): Number of classes used as types for noninherited attributes or parameters or local variables of methods in class *c*

$DAC\_t(c) = |DACSet\_t(c)|$

$DACSet\_t(c) = \{ \{T(a) \mid a \in A_I(c) \wedge T(a) \in C\} \cup \{T(v) \mid v \in Par(M(c)) \wedge T(v) \in C\} \cup \{T(w) \mid w \in Lv(M(c)) \wedge T(w) \in C\} \}$

Figure 5. Metrics definitions

As explained in the previous section, ISA generates subsystems and other entities that do not belong to any subsystem. In Figure 6 we define an algorithm for the object-based adaptation of objects and entities.

Given:

PSS = Set of primitive subsystems

UP = Set of unconnected programs

SP = Set of singular programs

UF = Set of unconnected files

CF = Set of common files

LF = Set of link files

P = Set of all programs in the system

F = Set of all files in the system

$F(s)$ = Set of files of subsystem s

$P(s)$ = Set of programs of subsystem s

$F_u(p)$ = Set of files that program p uses

$P_u(p)$ = Set of programs that program p calls

O = Set of all objects in the system

M(O) = Set of all methods (programs) in the system

A(O) = Set of all attributes (files) in the system

$M(o)$ = Set of methods of object o : $M(o) \subseteq M(O)$

$A(o)$ = Set of attributes of object o : $A(o) \subseteq A(O)$

$I(o)$ = Interface of object o : $I(o) \subseteq M(o)$

$A_u(m)$ = Set of attributes that method m accesses

$M_u(m)$ = Set of methods that method m invokes

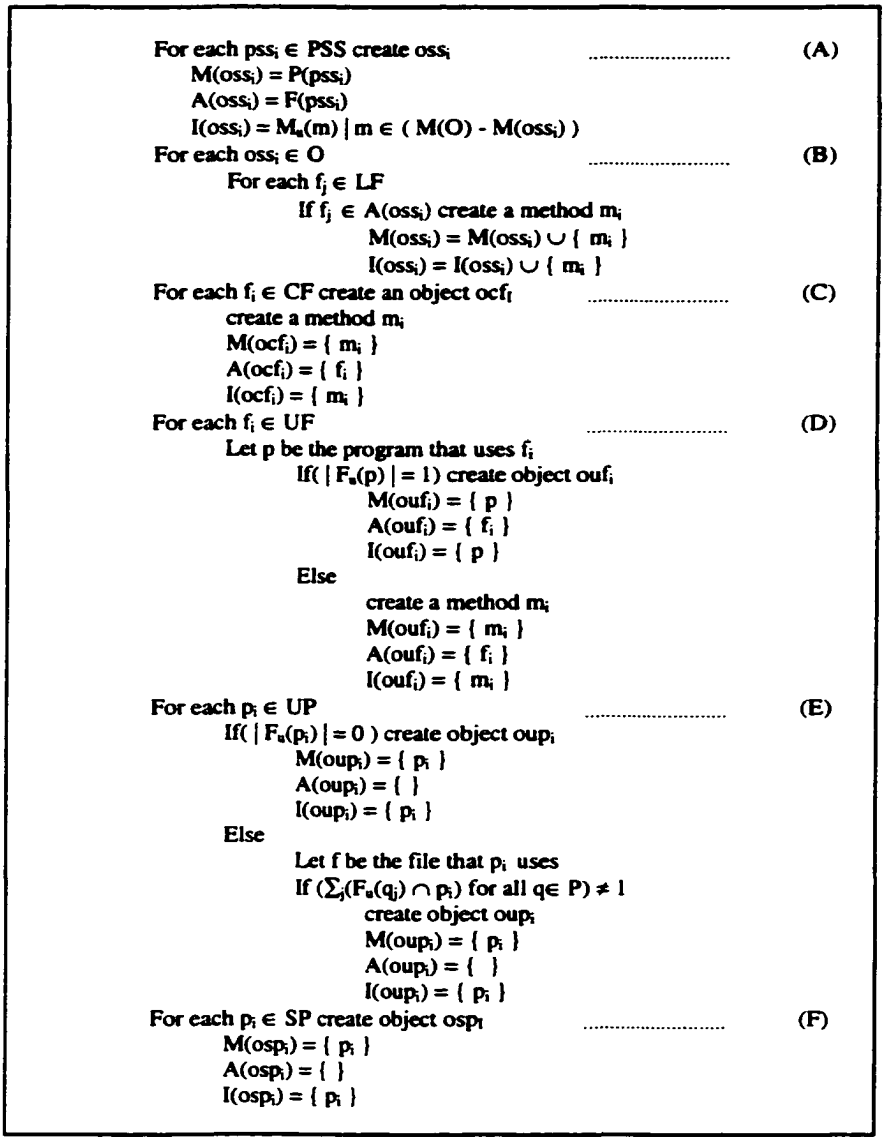

Figure 6. Object-based adaptation algorithm

An explanation of each section of the algorithm follows:

(A) We treat each primitive subsystem as an object, where the methods are the programs and the attributes are the files. The interface of this object includes the prototypes of the methods/programs that are called from other subsystems or external entities. We call these objects object-subsystems (OSS).

(B) Link files are files that belong to a subsystem but are also accessed (read/written) by programs in other subsystems. We create access methods (subroutines, functions) to access the link files. We replace the external access to link files by calls (object method invocations) to the new access methods of those link files. The interface of the modified object subsystems (OSS) now includes the prototypes of the access methods for the link files.

(C) Common files are files that are not assigned to any subsystem. As we did with link files, we create access methods for access to common files. Each common file with its corresponding access methods becomes an object. We call these objects object-common-files (OCF). We replace the access to common files by calls to the new access methods of those files. The interface of the OCF includes only the prototype of the access methods for the common file. In addition, files assigned to complex subsystems are treated as common files.

(D) Unconnected files are files that are used by only one program. If the program p using the unconnected file f uses only f, we create an object containing f and p. Then, we include the prototype of that program in the interface of the object. Otherwise, we create access methods for f and replace the access of f by calls to its access methods. We create an object containing f and its access methods. The interface of the object only includes the prototype of the access methods for f. We call these objects object-unconnected-files (OUF).

(E) Unconnected programs use fewer than two files. If an unconnected program p does not use any file, we create a pseudo-object consisting of code and no data (concept similar to global functions in hybrid object-oriented programming

languages). If *p* accesses one file *f*, then, we have two cases to consider. If no other program uses *f*, we have an unconnected program with an unconnected file. This case has already been discussed. Otherwise, we create a pseudo-object consisting of code and no data. In this case, we do not include *p* as a method of the object containing *f* since it may have semantic violations to the object. We call these objects object-unconnected-programs (OUP).

(F) Singular programs use at least two files, but are not assigned to any subsystem. This behavior suggests that such programs are not semantically related to any subsystem from the data cohesiveness point of view. For each singular program, we create a pseudo-object consisting of code and no data. We call these objects object-singular-programs (OSP).

Theorem 1: The object-based adaptation algorithm produces objects that contain data-cohesive methods (programs) and attributes (files).

Proof: The ISA subsystem decomposition produces a hierarchy of data cohesive subsystems and some other entities that do not fit into any specific subsystem. The programs and files in a subsystem are strongly related (i.e., files that belong to a subsystem are predominately used by programs in the same subsystem). The object-based adaptation algorithm produces encapsulation and information hiding by providing access methods to files that are accessed by programs in other subsystems and promotes data-cohesiveness by encapsulating entities (e.g., unconnected files and unconnected programs) that are strongly related. Therefore, the algorithm produces objects that contain data-cohesive methods and attributes. ∎

3.4. Component Creation: Wrapping and Interface Definition

Components are a set of objects wrapped together with an interface. The objective of the component creation phase is to wrap together data-cohesive OSS for non object-oriented systems and low-coupled classes for object-oriented systems. The interface of a component, which contains the services that the component offers, is the union of the interfaces of its objects.

Let C be the total set of components

O(c): set of objects that component c has

I(c): interface of component c where

$$I(c) = (\cup I(o)) \mid o \in O(c)$$

3.4.1 Component Creation of non Object-Oriented Systems

In this step, we wrap the objects produced during object adaptation (i.e., OSS, OCF, OUF, OUP, and OSP). We wrap objects with maximal locality of data and processing. In Figure 7 we define an algorithm to form components.

Given:

F = Set of files in OSS

P = Set of programs in OSS

$N_f(X)$ = Number of files in X, where X is a set of files

$N_p(X)$ = Number of programs in X, where X is a set of programs

k = Total number of OSS

B = Set of programs that access the file in OCF

G = Set of files that OSP access

H = Set of files that OUP access

M= Set of programs that OUP calls

L = Set of programs that P calls

Q = Set consisting of the (only) program in OUP

p = program that accesses the OUF

C= component containing program p

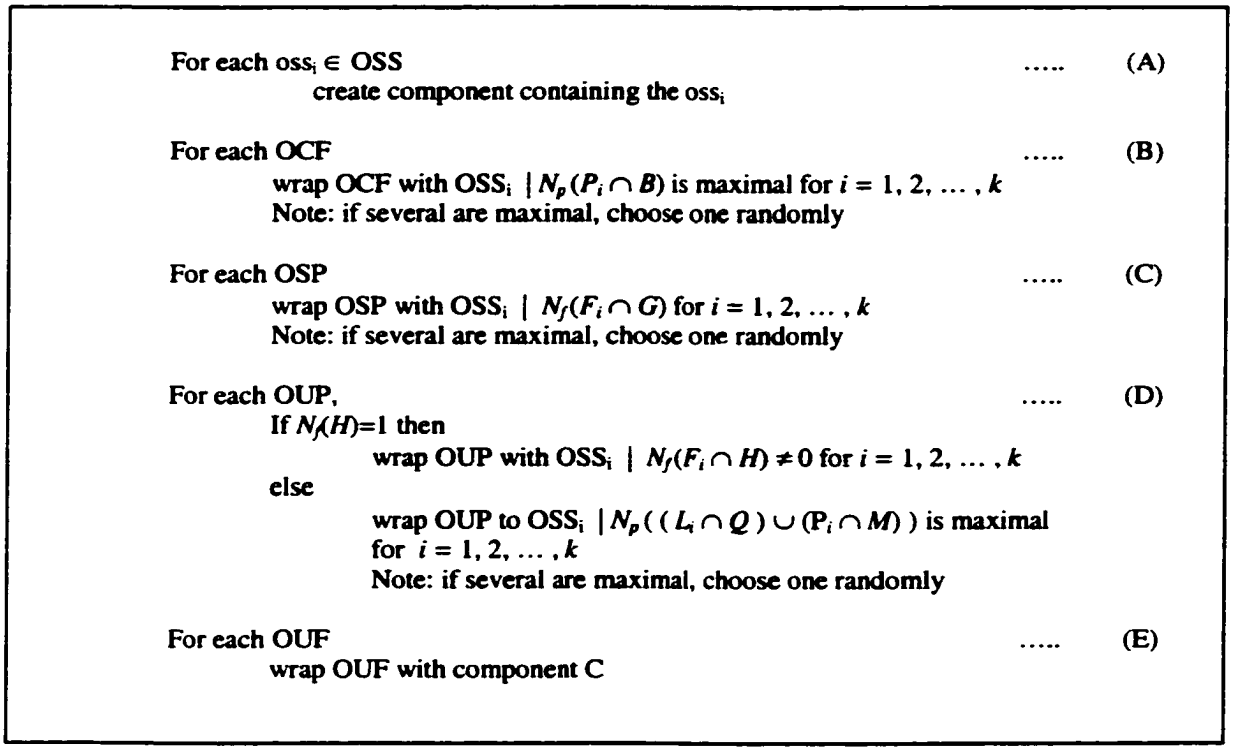

Figure 7. Component creation algorithm

An explanation of each section of the algorithm follows:

(A) Create a component that containing each of the object subsystems (OSS).

(B) Assign each object common file (OCF) to the component containing the OSS that has the maximum number of programs that use the common file in the OCF.

(C) Assign each object singular program (OSP) to the component that embodies the OSS that contains the maximum number of files that the singular program in the OSP accesses.

(D) For each object unconnected program (OUP) that has access to one file, wrap it with the OSS that contains the file. Otherwise, wrap the OUP with the OSS containing the programs to which the OUP has the highest call/called relationship.

(E) Assign each object unconnected file (OUF) to the component that contains the object with the program that accesses the file in the OUF.

Theorem 2: The component creation algorithm of non object-oriented systems produces components that contain data and processing cohesive objects.

Proof: The basic unit of the components are objects. Such objects contain data-cohesive methods and attributes. During the component creation, the algorithms wraps together processing objects (i.e., OSP, and OUP) with the objects that contain the attributes that they access. When the processing objects do not access any attribute (e.g., OUP) then the algorithm wraps the object with the object that has the highest call-called relationship. As a result, the algorithm produces components with processing cohesive objects. In addition, the algorithm wraps data objects (i.e., OCF and OUF) with the objects that have the methods that access them more. Therefore, the component creation algorithm for non object-oriented systems produces components that contain data and processing cohesive objects. ●

The component creation algorithm for non object-oriented systems produces components of different sizes. Based on the size of the component, several components may be joined into one component. We suggest wrapping the OSS with the highest data-cohesion. The hierarchical subsystem decomposition of the subject system performed by ISA and preserved during the object adaptation is highly suitable for merging components. We traverse the subsystem hierarchy and join components sharing the same parent in the hierarchical decomposition into a component until a stop condition, threshold, or constraint is reached. The maximum size of the component is an example of a stop condition. The joining of components in this way

assures high data locality because of the hierarchical data-cohesive feature of the decomposition process. The new interface of the component is the union of the interfaces of the objects that generated the component.

3.4.2 Component Creation of Object-Oriented Systems

The subsystem decomposition approach that we perform on the object-oriented system (using hierarchical clustering algorithms with formal descriptive features, direct-siblings links, and association coefficients similarity metrics) produces a hierarchical decomposition of the original object-oriented system. It resembles a tree where the nodes are classes (or maybe a strongly coupled set of classes) and the edges denote the similarity or distance (i.e., association coefficients) between the nodes.

For the component creation, we want a partition of the hierarchy instead of a hierarchy of clusters. We get the components by pruning the hierarchy at an appropriate level and by considering only the top-most clusters as a single component [Anqu99]. Cutting at height 0 produces components with only singleton clusters. If we cut at the maximum height, the entire system is a single cluster and becomes a single component. The heuristics that we follow cut the hierarchy at different heights based on the size of the component (i.e., the number of classes). Each top-most cluster becomes a component. Each component consists of a set of classes, and the interface of the component is the union of the interfaces of its classes.

3.5. Allocation of Components

The objective in this step is to allocate the components to one or more sites or processing elements. During allocation, one or more components may be allocated to the same site. We allocate components with maximal locality of data and processing.

The allocation of components to sites plays an important role in the efficient implementation on a distributed environment. For example, assume that we have a network with many sites, each site being a network of workstations or LAN. We can allocate a set of components belonging to the same hierarchy sub-tree to a LAN because of the insignificant communication cost/delay. We can assign each of the components to different computers in the LAN. Components that belong to different hierarchy sub-trees can be allocated to different nodes in a more geographically dispersed network.

In this research, we adapt the technique proposed by Bastarrica for the efficient allocation of components to different sites in the network [Bast98].

For our model to be correct, the following four conditions must hold:

1. Completeness: Each component is allocated to one and only one node in the network
2. Storage: The total storage required by all the components allocated to each node does not exceed the node's storage capacity.
3. Connectors: A component in one node communicates with a component in another site only through the network connectors.
4. Bandwidth: For each connector, the total communication bandwidth does not exceed the connector's capacity.

To address efficient distribution of components over the network, we need knowledge of both, the network, and the system.

Assume that we have C components and N nodes.

- Create an array NS with the network nodes' storage capacities

NS(i) = node storage capacity of the i_{th} node, i = 1..N

- Create an array *CS* with the storage required by components

  CS(i) = components storage requirement of the i_{th} component, i = 1..C

- Create a matrix *NConB* with the bandwidth communication between nodes in the network

  NconB(i,j) = communication bandwidth between i_{th} and j_{th} nodes, i = 1..N, j = 1..N

- Create a matrix C*ConB* with the required connection bandwidth between components. This matrix is based on the amount of communication between the classes of the two components.

  CconB(i,j) = required bandwidth between i_{th} and j_{th} components, i = 1..C, j = 1..C

To achieve an efficient allocation of components, we use Binary Integer (linear) Programming (BIP) algorithms [Zion74]. A Linear Program (LP) is a problem that can be expressed as follows [Four99]:

Minimize or maximize cx
subject to Ax = b
x >= 0

Where x is the vector of variables to be solved for, A is a matrix of known coefficients, and c and b are vectors of known coefficients. The expression "cx" is called the objective function, and the equations "Ax=b" are called the constraints. All these entities must have consistent dimensions. Usually A has more columns than rows, and Ax=b is therefore quite likely to be under-determined, leaving great latitude in the choice of x with which to optimize cx.

LP problems are characterized by objective functions and constraints where each incremental unit of a variable of the problem contributes the same amount to the value of the objective, and consumes the same amount of resources used in producing that variable. Linear integer programming (LIP) problems are essentially the same, but some or all the variables are restricted to integral values. BIP (also called zero-one integer programming) problems are a special case of LIP problems, where the variables are restricted to 0 and 1 values.

We want to create a BIP model that minimizes remote communication. Let x be a basic decision variable, and y and auxiliary decision variable:

$x_{i,j}$ = 1 if component i is assigned to node j, 0 otherwise

$Y_{a,i,b,j}$=1 if component a is assigned to node i and component b is assigned to node j, 0 otherwise

Notice that $Y_{a,i,b,j} = x_{a,i} * x_{b,j}$ and

$Y_{a,i,b,j} <= x_{a,i}$

$Y_{a,i,b,j} <= x_{b,j}$

The objective function is:

Minimize $Z = \Sigma_{i=1\ to\ N} \Sigma_{j=1\ to\ N,\ j \neq i} \Sigma_{a=1\ to\ C} \Sigma_{b=1\ to\ C}$ ($Y_{a,i,b,j}$ * CConB(a,b))

subject to

$\Sigma_{j=1\ to\ N} (x_{i,j}) = 1$ ..... (1)

$\Sigma_{i=1\ to\ C} (x_{i,j} * CS(i)) <= NS(j)$ ..... (2)

$\Sigma_{a=1\ to\ C} \Sigma_{b=1\ to\ C}$ ($Y_{a,i,b,j}$ * CConB(a,b)) <= NconB(i,j) ... (3)

Constraint (1) deals with the completeness condition, (2) with the storage condition, and (3) with the connectors and bandwidth conditions.

There are many solvers available (e.g., [Lind99] and [GAMS99]) that employ the branch-and-bound algorithm and similar techniques to solve this kind of problems.

Optimal allocation of components in the network includes many issues, such as types of the processing units, storage capacity, network bandwidth, and special processing requirements of components (e.g., databases and numeric processors). Optimal allocation requires a model that optimizes a set of criteria such as cost, availability, and storage.

In this research, efficient allocation of components is defined as the allocation of components to different sites that minimizes the inter-site communication between the components. This research achieves efficient allocation of components by focusing on storage and communication, both needed by components and offered by the network and processing elements. There is a significant amount of active research dealing with the problem of optimal allocation of components in a distributed network of processors. For related research on this problem, see Purao and Bastarrica [Bast98], [Pura98b].

3.6. Middleware and Distributed Systems Implementation

At this stage, the remaining step for the implementation of the system as a distributed system is the use of a middleware technology (such as CORBA or DCOM) that allows the interconnection/interfacing of components. The middleware technology we use is CORBA.

CORBA is a set of specifications for providing interoperability and portability to distributed object-oriented applications. CORBA-compliant applications can communicate with each other regardless of location, implementation language,

underlying operating system and hardware architecture. Communication mainly occurs through method invocations from a client object on a server object [Vasu97]. There are several commercial and free providers offering CORBA, including IONA, HP, IBM, SUN, Xerox, and Univ. Colorado. Different platforms and programming languages are supported.

In order to make requests on a server object, the client must know the types of services provided by the server object and the mechanism to invoke those services. The server object interface specifies the types of operations it supports and defines how to invoke those operations. These interfaces are defined using the OMG Interface Definition Language (IDL). IDL defined methods can be written in and invoked from any language that provides CORBA bindings (C, C++, Smalltalk, COBOL, and Java at present). It basically acts as an intermediate neutral interface, which allows client and server objects written in different languages to inter-operate across networks and operating systems. Another element of the CORBA architecture is the Object Request Broker (ORB) that provides the underlying communication framework that handles the transparent interaction between the client and server objects. Figure 8 shows the basic CORBA architecture.

The last step is mapping the component interfaces to IDL. First, we code the interfaces in IDL. Then, the IDL compiler translates clauses into fragments of the underlying (legacy system) language such as COBOL, C++, and JAVA. The ORB manages the interaction and communication between components.

Finally, we stress the importance of the evolutionary migration approach. As shown in Figure 2, once we have created the components, reengineering becomes

more feasible since components can be addressed independently for maintenance, reuse, and testing. In addition, if a middleware technology is used, each component can be considered for migration to a different hardware platform and programming language.

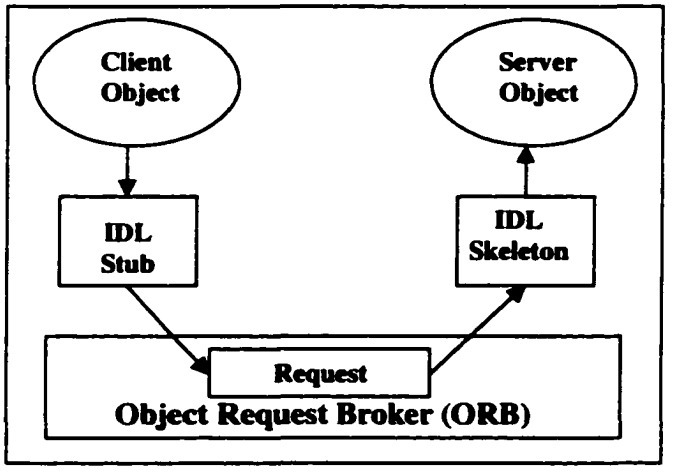

Figure 8. Basic CORBA architecture

CHAPTER 4. THE REENGINEERING ENVIRONMENT

Based on the reengineering methodology, we now define an environment which implements the methodology. Methodologies and techniques for reengineering of legacy systems cannot be applied without tool support on even medium sized systems. Tools help to cope with the vast amount of information normally found in legacy systems. They can provide a developer with different views of a system, point to possible problems in the code, and help improve the software accordingly. We refer to the methodology together with the integrated tool set as the reengineering environment.

During the development of this research, we created some primitive and prototype tools to support the reengineering and migration tasks. For the specific goals of the research, we needed tools for parsing, design recovery, code analysis, data mining, and clustering. Since the task of creating a framework with these tools is a major work, we conducted a survey of existing tools. We also relied on surveys of tools reported in the literature [Gann99], [Amst98]. Most tools deal with one specific task (e.g., parsing, metrics, data mining, or clustering).

We evaluated commercial industrial tools that are very mature. We also evaluated research tools. We evaluated the tools with publicly available source code of applications and frameworks, allowing us to compare the tools with the same code. After significant experiments, we chose a set of tools that we integrated to implement the methodology.

Some of the tools that we used (e.g., SNiff+, CodeCrawler, Concerto/Audit, and Goose) in this research provide output and accept input in the standard interface CDIF

(Case Data Interchange Format) [EIA 99]. CDIF is an industrial standard for transferring models created with different tools. In addition, some of the tools (e.g., CodeCrawler, Concerto/Audit, and Goose) share a language-independent model for information exchange called the FAMOOS Information Exchange Model (FAMIX) [Bar99]. The FAMIX model provides a language-independent representation of object-oriented source code (e.g., C++, JAVA, Smalltalk, and Ada) and is used as a basis for exchanging information about object-oriented software systems [Deme99c].

In this chapter, we describe the set of tools that we chose for the integrated reengineering environment. We classify the tools in the environment in different categories such as environment and system tools, reverse engineering tools, data mining tools, and metric tools. Sections 4.1 - 4.4 cover the different classes of tools. Section 4.5 describes the integrated reengineering environment.

4.1. Environment and System Tools

Some of the tools need specific operating environment and system tools. Such operating environments offer software engineering capabilities such as configuration management and version control.

4.1.1 VisualWorks Smalltalk

VisualWorks Smalltalk [CIMC99] is the Smalltalk implementation from Cincom Systems, formerly from ObjectShare and PARC. Some of the tools that we describe later use VisualWorks Smalltalk as its operating environment (e.g., CodeCrawler).

4.1.2 ENVY

ENVY [OTI99] is a set of development tools for Smalltalk from Object Technology International Inc. ENVY provides facilities required for the development

and maintenance of large software systems. It provides functions for configuration management, version control, change management, component management, and history management.

4.2. Reverse Engineering Tools

We used different tools that provided different views and capabilities for analysis of the source code. Reverse engineering tools are the heart of any reengineering or maintenance activity. Understanding the legacy code is one of the most crucial, complex, and time consuming activities in the maintenance and reengineering of legacy systems.

The methodology that we propose is not totally automatic; it needs human expert interaction to generate a decomposition and distribution of the legacy system. Consequently, the human expert needs to have a clear understanding of the architecture of the legacy system.

We selected SNiff+, Tablegen, and Goose to analyze the code of legacy systems.

4.2.1 SNiff+

SNiff+ is a set of source code engineering tools and services (from TakeFive software) that help develop and maintain large software systems [Take99]. SNiff+ is an open, extensible and scalable source code engineering tool for C, C++, FORTRAN, Java™, CORBA IDL and other languages. SNiff+ supports the following tasks within the development and maintenance process: code analysis, browsing and comprehension, project and code management for teams, source code editing, documentation building, version and configuration management, build management, and debugging support.

In this research, the feature that we use SNiff+ for is the parsing of code of C++ and JAVA programs.

To show the functionality of SNiff++, we parsed and analyzed the code of the Swing component from JAVA JDK 2.0 (From Sun Microsystems). The Swing components are written in Java, without window-system-specific code. Swing facilitates a customizable look and feel without relying on the native windowing system, and simplifies the deployment of applications. Swing also supports a pluggable look and feel architecture. This feature of swing gives users the ability to switch the look and feel of an application without restarting it and without the developer having to subclass the entire component set [Sun 99].

We parsed the source code of JAVA-Swing with SNiff++ and extracted information about the different entities (e.g., classes, methods, and instance variables) and relationships (e.g., method invocation and class aggregation) useful for the analysis of the code. We retrieved information of 1637 classes, 4361 instance variables, and 11820 methods. Figure 9 shows an output summary of statistics that SNiff+ extracted from the Swing code. The figure also shows information about the number of include files.

SNiff++ also provides different views of the code, such as hierarchical class browsing, symbol browser, class browser, cross referencer, and include browser. Figure 10 shows an example of one of the hierarchical class views that we obtained from the code. The figure shows a screen shot showing the inheritance tree of class hierarchies.

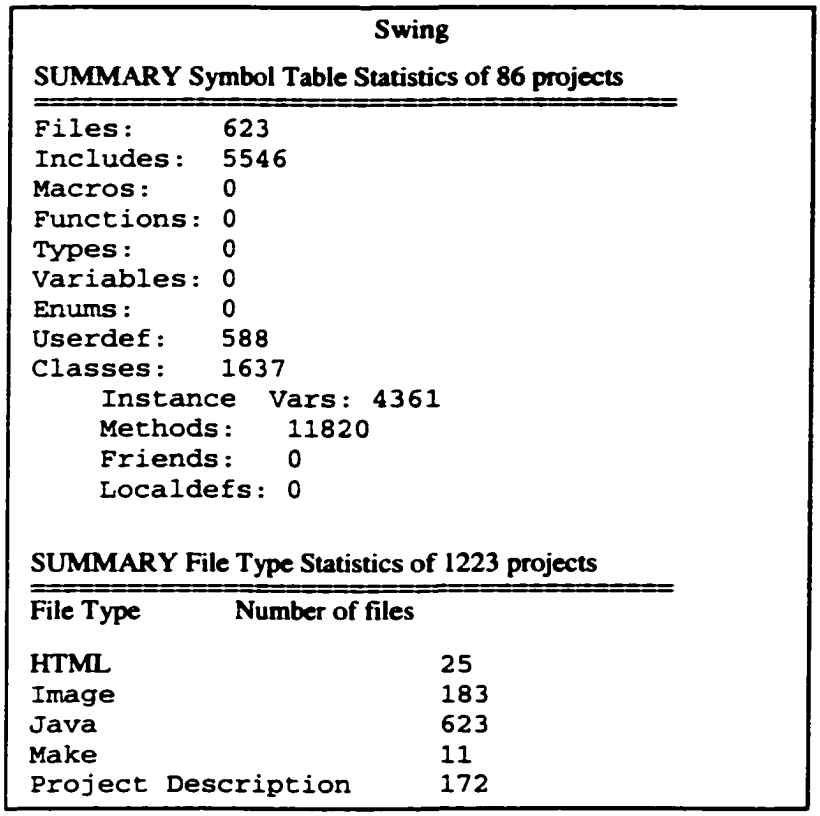

Figure 9. Statistics of the JAVA-Swing code

4.2.2 Tablegen

Tablegen (Table Generator) [Sass99] is a parser for C++ systems. It extracts design information, such as accesses to variables, method invocations, and variable declarations. Tablegen generates tables with information about classes, variable accesses, method invocations, and inheritance.

Often, the only reliable documentation of a legacy system is the source code itself. The quality of the parser of the reengineering tool is therefore of utmost importance. Pieces of code that are not parsed correctly cannot be reengineered. The TableGen parser saves relevant data about the sources in tables and as a FAMIX/CDIF file. The tables can later be queried by programs written in the CQL-language, a

dialect of the SQL query language. Tablegen also produces output in the FAMIX/CDIF format that may be used as an interface to transfer information to other tools [Bar99].

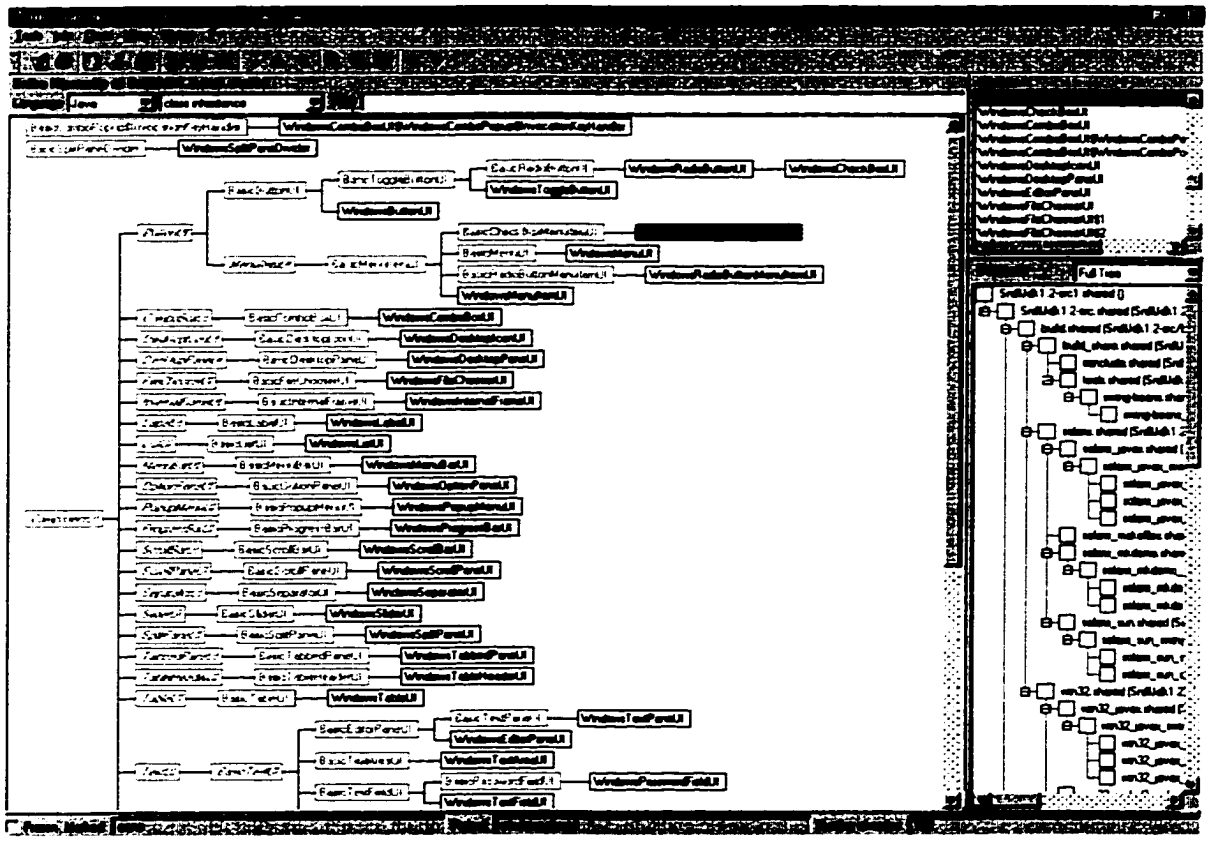

Figure 10. Swing hierarchy tree

4.2.3 Goose

Goose is a collection of tools for analyzing the design of object-oriented software systems [Bar98]. It extracts the basic entities of object-oriented systems (i.e., classes, methods and attributes) and their relationships (e.g., inheritance, aggregation, and method invocation). Using the design information, it provides many ways to visualize the design as a graph, to create abstract views on the design, and to compute further information about the design like object-oriented metrics or potential design flaws.

Goose integrates tools for parsing, metrics, and visualization (e.g., SNiff+, Tablegen [Sass99], and Graphlet [Pass99]) to provide its functionality.

Goose extracts the design information from the source files of legacy systems. The result of the code analysis is a repository file containing the extracted design information in "simple format" (SF). The SF consists of several text tables that keep information about classes, instance variables, methods, instance variable accesses, method invocations, and class inheritance.

Goose allows visualization of the underlying legacy system as a graph. Figure 11 shows a graph of a large system as displayed by Goose. In the graph, the nodes correspond to object-oriented entities, and the edges correspond to their relationships.

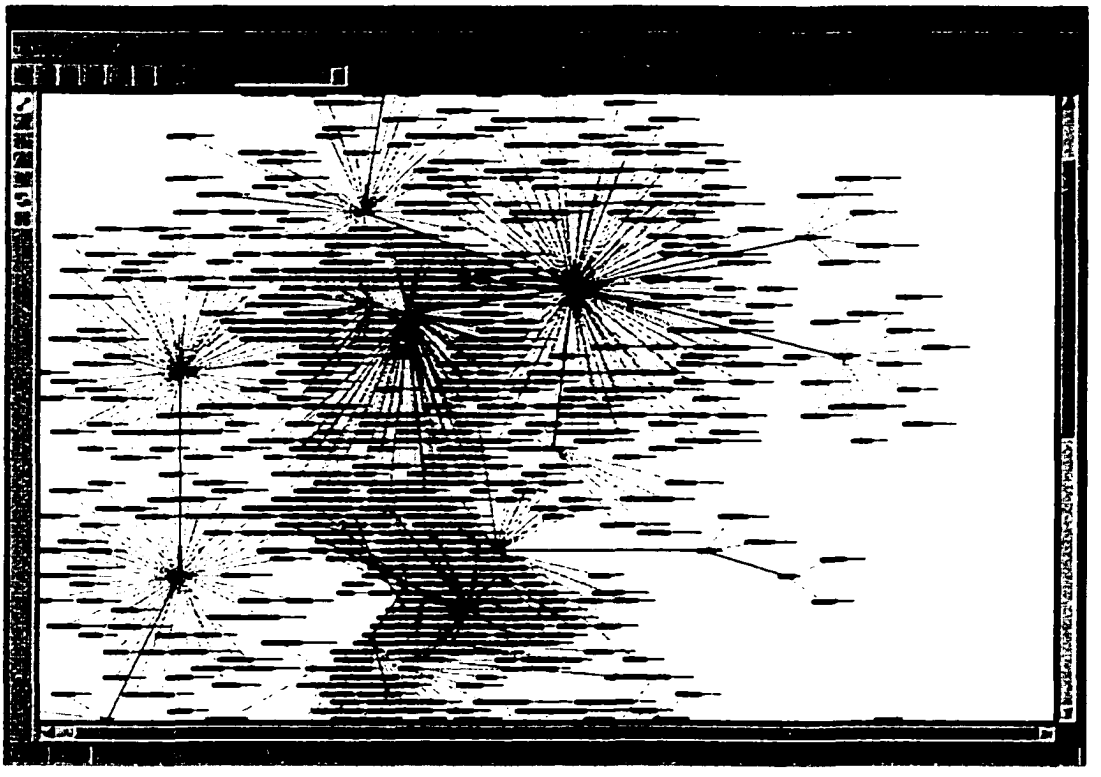

Figure 11. Goose visualization of legacy system

Goose also supports the computation of object-oriented metrics such as DAC, DIT, NOC, WMC, and RFC. It also allows querying and graph manipulation (e.g., collapse, delete, add, and filter nodes).

4.3. Data Mining Tools

Data mining is the analysis of large amounts of data to discover relationships and patterns that have not previously been known. In the area of software engineering the use of data mining is a relatively new but promising area to discover hidden relationships and patterns in the code. Data mining has been used in other contexts such as basket analysis, market demographic clustering, and prediction of sales revenues. Some industrial vendors are offering business solutions suites for data mining and knowledge discovery. These vendors include Information Discovery Inc. [Pass99], Data Distilleries [Data99], Oracle [Orac99], and IBM [IBM00], [IBM 98].

In Sections 4.3.1 and 4.3.2 we describe two data mining tools that we used in this research, Intelligent Miner and RE-ISA respectively.

4.3.1 Intelligent Miner

Intelligent Miner (IM) is a suite of statistical, processing, and mining functions for analyzing large amounts of data. IM is one of the Business Intelligence Products offered by IBM [IBM00], [IBM 98].

IM provides association, clustering, sequential patterns, classification, and prediction data mining functions. IM also provides visualization tools for viewing and interpreting the mining results. The graphical representation of the output is very useful for analyzing large data sets. Figure 12 shows an output of the visual

representation of the demographic clustering function. The multiple rows of graphs are designed to give the user an understanding of the clustering described in the result.

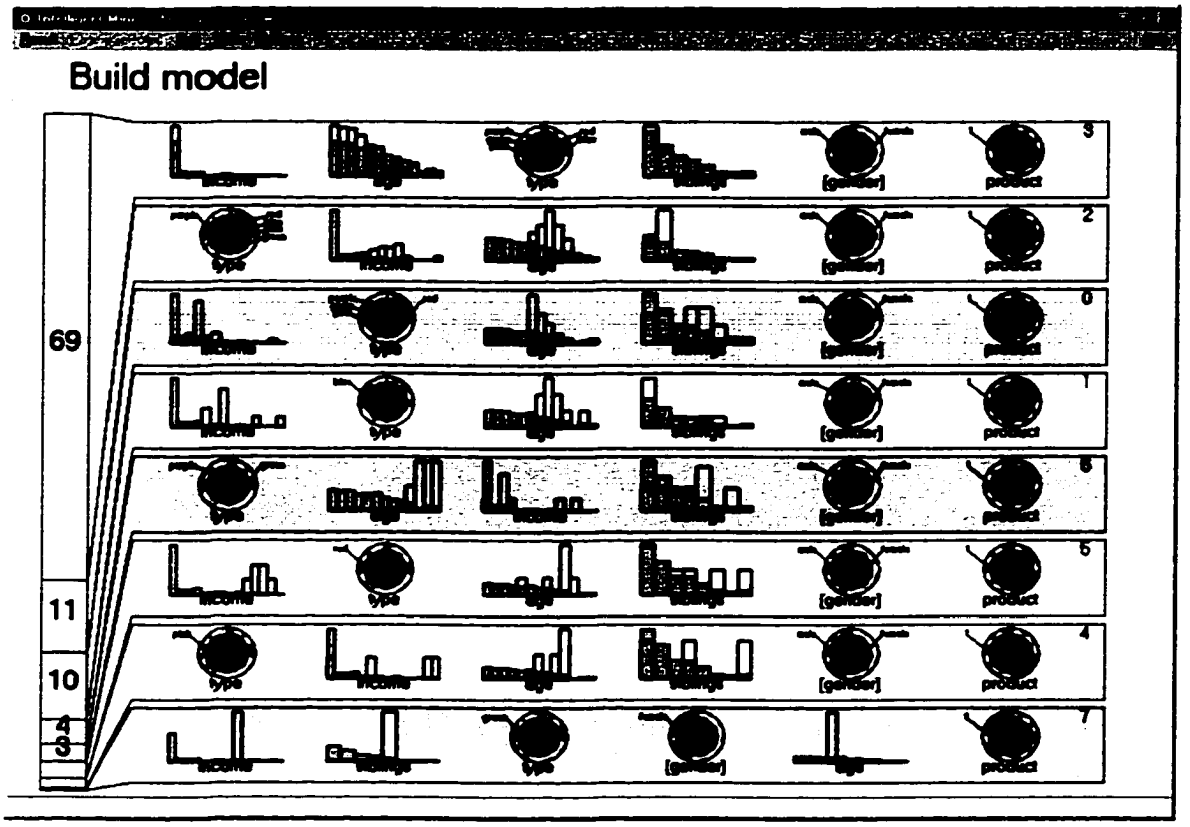

Figure 12. Visual representation of demographic clustering

In our research, we use associations rules to drive the data cohesive hierarchical subsystem decomposition of legacy system. IM provides us with association rules mining functions. Association rules are used to find items in a transaction that imply the presence of other items in the same transaction (e.g. when a *customer buys* diapers, then the *customer* also *buys* eggs in 90% of cases). We use association rules to discover association in legacy systems (e.g. when a *program uses* File1, then *the program* also *uses* File2 in 83% of cases). Figure 13 shows an example of the IM output for finding such kind of associations rules in legacy systems.

4.3.2 RE-ISA

RE-ISA is the tool that implements the ISA methodology explained in Section 3.2.1 [Mont98a]. RE-ISA is a system-level tool that decomposes a software system into a hierarchy of data cohesive subsystems. RE-ISA follows three steps to identify subsystems, (1) Build a database view of the system, (2) Perform data mining, and (3) Consolidate and interpret results.

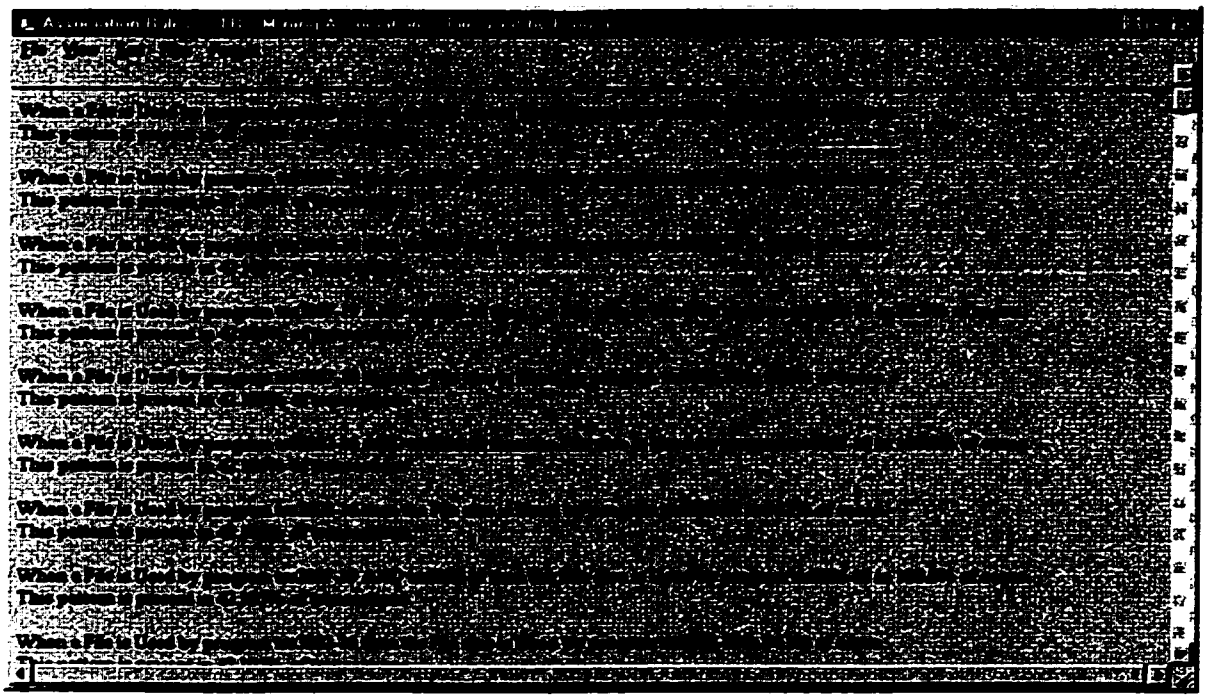

Figure 13. Intelligent Miner association rules

The major features of RE-ISA are (1) RE-ISA is written in JAVA and is platform independent, (2) RE-ISA uses a graphical user interface (GUI) to accept user input, (3) The output of RE-ISA is a textual representation, (4) RE-ISA executes each one of the major steps of ISA independently, (5) RE-ISA can accept information generated with other tools (e.g., parsers) as input to perform the data mining or clustering, and (6) RE-ISA can process large systems. Figure 14 shows a screen shot of a session with the

RE-ISA tool. The upper left pane in the figure allows the user to specify the source code to mine. The bottom left pane allows the specification of parameters for the mining functions. The right pane shows an example of the textual output of a subsystem decomposition that shows two subsystems (i.e., S1, and S2).

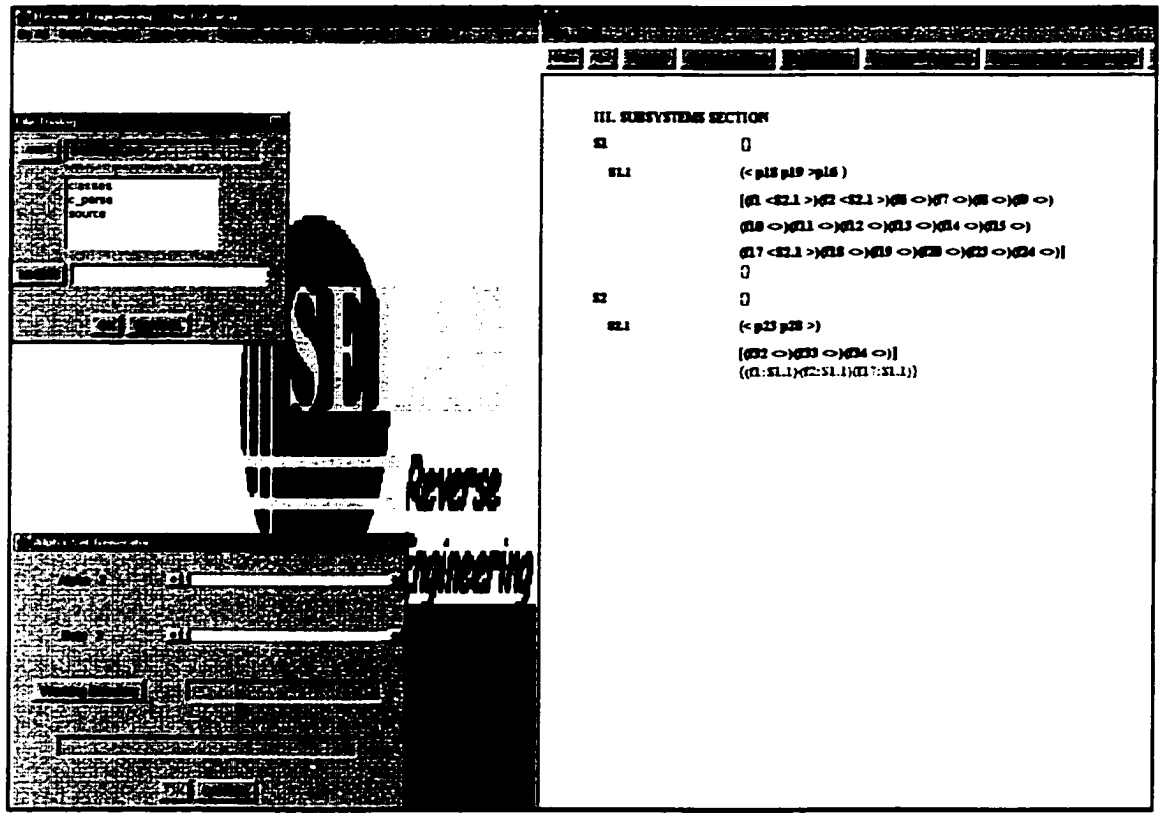

Figure 14. RE-ISA graphical user interface

4.4. Metric Tools

The research community has worked actively to define metrics for the non object-oriented and object-oriented paradigms. At the same time, several tools have been developed to generate software metrics [Chid94], [Deme99b], [Hend96].

In this section we describe the tools that we use in this research that provide us with information about measurements (metrics) of legacy systems.

4.4.1 CodeCrawler

CodeCrawler is a language-independent reverse engineering tool which combine graphs and metrics to generate views of object-oriented systems [Lanz99]. CodeCrawler supports reverse engineering of large object-oriented projects. It combines the immediate appeal of visualization with the scalability of metrics. Furthermore, it allows the user to tailor what information is presented as well as how it is presented. CodeCrawler is a tool that encompasses both graph visualization and metrics combined in a simple approach where (a) the graph layout is very simple and (b) the extracted metrics are straightforward to compute [Deme99b], [Bar99].

CodeCrawler enriches a simple graph like tree with metric information of the object-oriented entities it represents. In a two-dimensional graph it renders up to five metrics on a single node at the same time. The five metrics are possible by making use of the position of the node (X-Y coordinates of the node can render two measurements), node size (the width and height of a node can render two measurements), and color of node (gray tones, where darker means higher values). Table 4 shows the metrics that CodeCrawler supports. The first column has the name of the metric, and the second column contains a description of the metric.

CodeCrawler works in the VisualWorks Smalltalk + Envy environments. CodeCrawler uses the facilities provided by the VisualWorks environment for the Smalltalk code parsing. For other languages like C++ and Java, it relies on SNiff+ to generate code representation using the CDIF standard and the FAMIX Model [Deme99c].

Table 4. Metrics supported by CodeCrawler

Metric Name	Description
AHNL	Deep in the inheritance tree where the attribute ExtendSuper
HNL	Hierarchy nesting level: deep of the class into the inheritance tree
LOC	Number of lines of a method
MCX	Method of a method
MHNL	Level of dept of inheritance of a method
MSG	Number of messages of a method
NA	Number of accessors of a class
NAA	Number of attribute accesses of an attribute
NAM	Number of abstract m ethods
NC	Number of constructors of a class
NCM	Number of classes having methods that access an attribute
NCV	Number of class variable of a class
NGA	Number of times an attribute is accessed by methods non-local to its class
NI	Number of method invocation of a method
NIA	Number of inherited attribute of a class
NIV	Number of instance variable of a class
NLA	Number of times an attribute is accessed by methods local to its class
NM	Number of methods accessing an attribute
NMA	Number of methods added of a class
NMAA	Number of attribute access of a method
NME	Number of Method Extension of a class.
NMI	Number of methods inherited of a class
NMO	Number of methods overridden of a class
NOC	Number of immediate children of a class
NOM	Number of methods of a class
NOMP	Number of Method Categories of a class.
NOS	Number of statements of a method
NTIG	Number of times a method is invoked by methods non-local to its class
NTIL	Number of times a method is invoked by methods local to its class
PriA	Number of private attributes of a class
PriM	Number of private methods of a class
ProA	Number of protected attributes of a class
ProM	Number of protected methods of a class
SIX	SIX number for a class
WLOC	Number of lines of all the methods of a class
WMCX	Sum of all the methods complexity metrics.
WMSG	Number of messages of all the methods of a class
WNAA	Total number of attribute accesses computed per access a class
WNI	Total number of method invocations of a class
WNMAA	Total number of attribute accesses computed per method a class
WNOC	Total number of children of a class
WNOS	Number of statements of all the methods of a class

CodeCrawler allows visualization in different types of graphs (e.g., stapled, histogram, checker, correlation, inheritance tree, circle, and confrontation). The nodes are object-oriented entities (i.e., classes, methods, and attributes), and the edges are relationships between the entities (e.g., class inheritance, method invocation, and attribute access). As explained before, it is possible to visualize five metrics in one single graph. Depending on the kind of nodes selected, it is possible to select different relationships and metrics. CodeCrawler has a repository facility to store graph definitions. Figure 15 shows the different screens dialogs that allow us to define the graph. The upper left dialog allows for the selection of the type of graph. The upper

right dialog specifies the metrics to consider. The lower left dialog specifies options for the graphical display of the graph. The lower right dialog shows the graph repository that allows the selection from a set of predefined graphs.

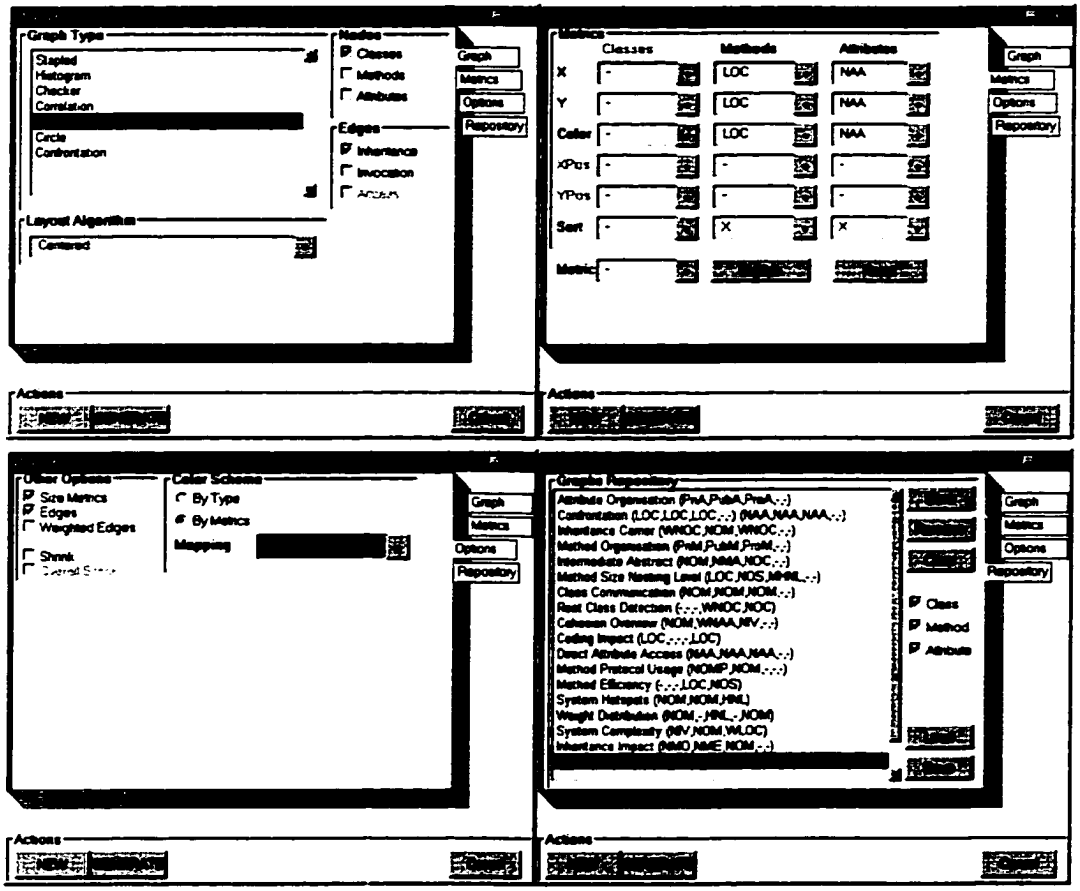

Figure 15. Graph definition dialogs

CodeCrawler also allows the user to select the model (i.e., which classes, methods, and attributes) that should be considered for the analysis and visualization. It is also possible to include information recovered with other tools (e.g., SNiff+) through the CDIF interface. Figure 16 shows the dialog that allows the definition of the model. In the different panes, the user can select which classes, methods, and attributes he wants to include in the model for future analysis and visualization.

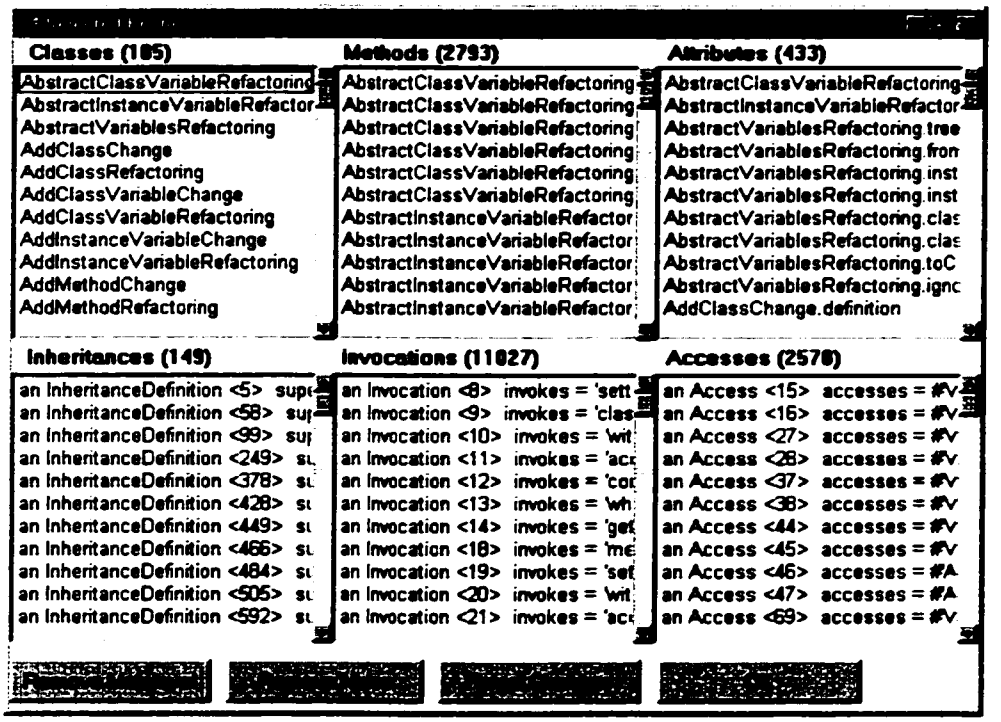

Figure 16. Model definition dialog

The CodeCrawler visualization capabilities, which provide metrics and relationships, make it a suitable tool for code analysis and program comprehension. We used CodeCrawler to analyze and visualize the Refactoring Browser [Bran99] SmallTalk application. The Refactoring Browser is an advanced browser for VisualWorks, VisualWorks/ENVY, and IBM Smalltalk. It includes all the features of the standard browsers plus several enhancements. Figure 17 shows an example of an inheritance graph (system complexity graph) of the Refactoring Browser. In this graph the nodes are classes and the edges represent inheritance. The Refactoring Browser has 177 classes, and 149 inheritance relations. The metrics considered are NIV, NOM, and WLOC corresponding to the width, height, and color of the node respectively. In the graph, each box represents a class. A node connected to another node means that the class in the bottom inherits from the class in the top. The width of the node

provides information on how many instance variables a class has. The height of the node provides information on how many methods a class has. The color of a node provides information about the number of lines of all methods of a class.

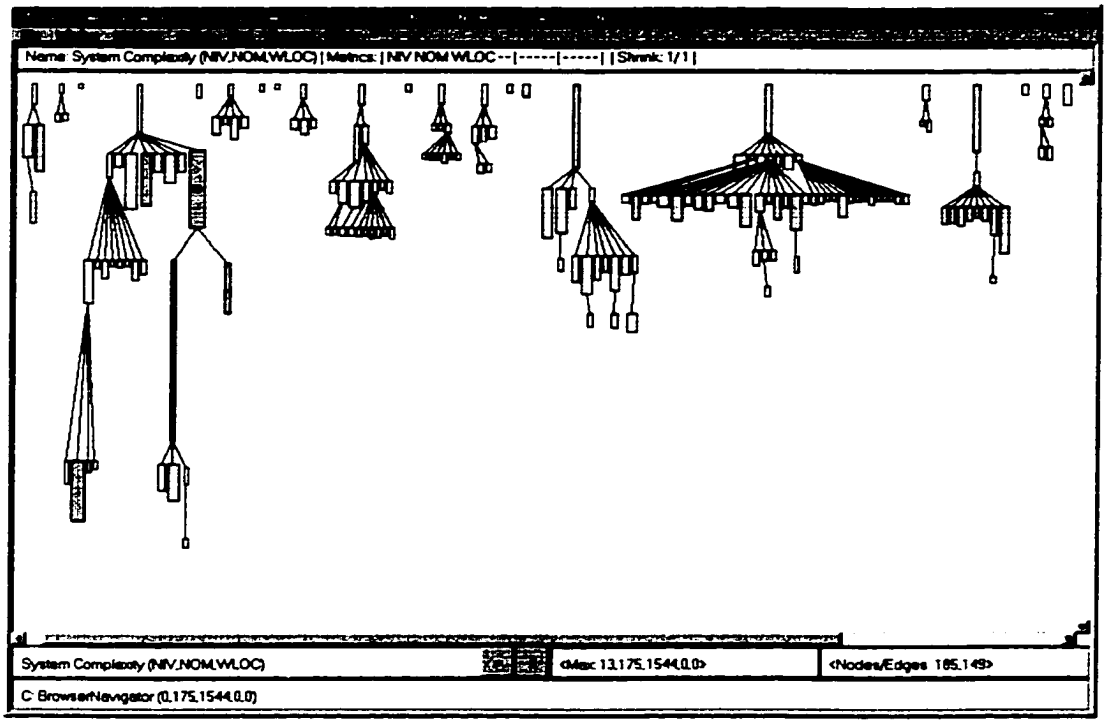

Figure 17. Refactoring Browser inheritance graph

CodeCrawler provides a number of other features that greatly enhance reverse engineering activities such as the query of the graph to identify a node according to some criteria and code navigation via the graph. Each graph entity is linked to the code entity that it represents, so the reverse engineer can browse the code related to the displayed entity as well as its metrics.

There are many possible combinations of graphs layouts, metrics, nodes, and edges in which CodeCrawler can be used. Figure 18 shows a confrontation graph focusing on the RefactoringBrowser class. CodeCrawler displays the information of

the current displayed graph (top border) and the information related to the entity under which is the mouse (bottom border). The graph allows us to understand the cohesion of the class by looking at the way methods access instance variables. Edges in the confrontation graph represent an instance variable access by methods. In the graph, the instance variables are the nodes in the top, and the methods are the nodes in the bottom. The RefactoringBrowser class has 5 instance variables, 48 methods, and 85 variable accesses. The metric considered for methods was LOC and for variables was NAA.

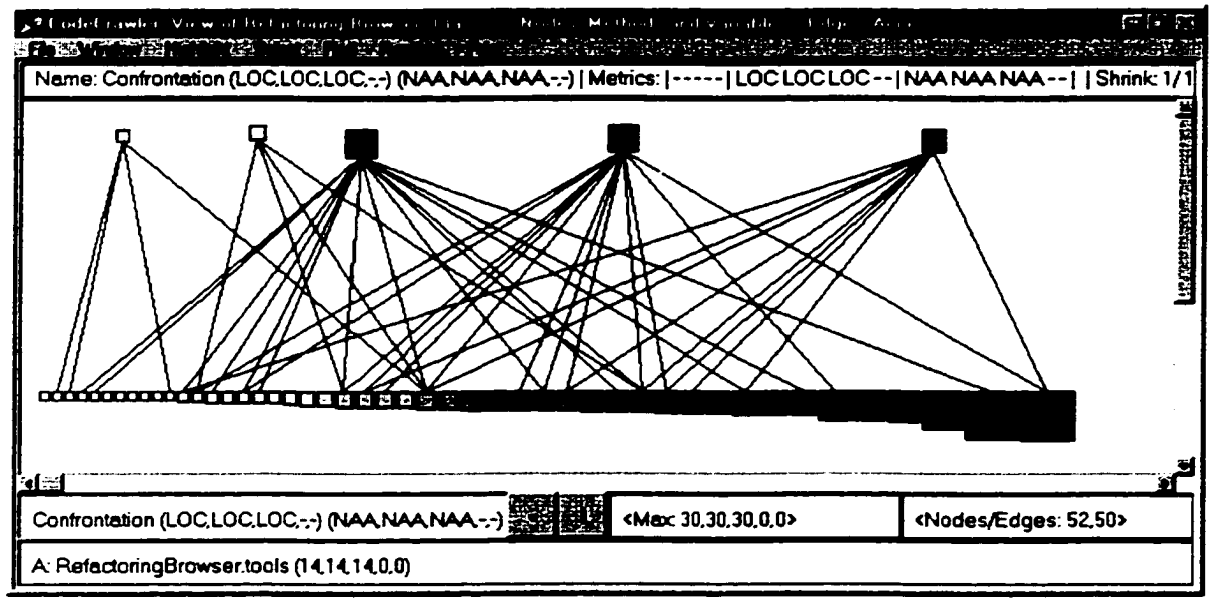

Figure 18. RefactoringBrowser class confrontation graph

Code crawler can be used for different purposes such as (1) Assessment of System Complexity (big classes, small classes, large inheritance hierarchies, standalone classes), (2) Method Efficiency Correlation (overlong methods, short methods, badly formatted methods, inefficient methods, empty methods), (3) Service Class Detection (service classes, classes with overlong methods), (4) System Hotspots

(big and small classes in the system), (5) Method Size Histogram (overlong methods, short methods,), (6) Direct Attribute Access (overused and never used attributes), (7) Class Communication (heavily communicating classes), and (8) Confrontation (apply on one single class and look at its internal details) [Deme99b], [Lanz99]. The right bottom screen shot of Figure 15 showed the repository with examples of graph definitions for these different purposes.

4.4.2 Concerto2/Audit-RE

Concerto2/Audit-RE (Audit-RE) provides all the features of the Concerto/Audit [Sema98] tool and the best-practice heuristics for reengineering of object-oriented code proposed in the FAMOOS project [Famo99]. Audit-RE parses the code using the Audit parser and Tablegen. Audit-RE provides different views of the source code, automatic violation of best-practice heuristics, and object-oriented metrics.

Audit-RE provides several synchronized views of the source code. The Application View supplies the list of files being parsed, the Module View displays the source code, the Query View shows the results of queries about the source code, and the Graphical View shows the different relations that exist between parts of the source code in a graphical way. All these views are synchronized, the selection of an item in a view will immediately update the corresponding parts in the other views. These views provide a good basis for browsing the code, and understanding the relations between the different objects, and the outcome of certain metrics and heuristics [Bar99].

Figure 19 shows a screen shot of a session with Audit-RE. It shows the inheritance relationship. The module view that shows the code is on the left. The query view, which is in the middle, gives textual information about the parent-child

classes. The application map (graphical view) is shown on the right. If we click on a node in this inheritance graph or we click a row in the query view, we can inspect the implementation of that class in the Module view. In the lower left-hand side of the figure, we can see the module view which shows all the modules (files) considered for analysis.

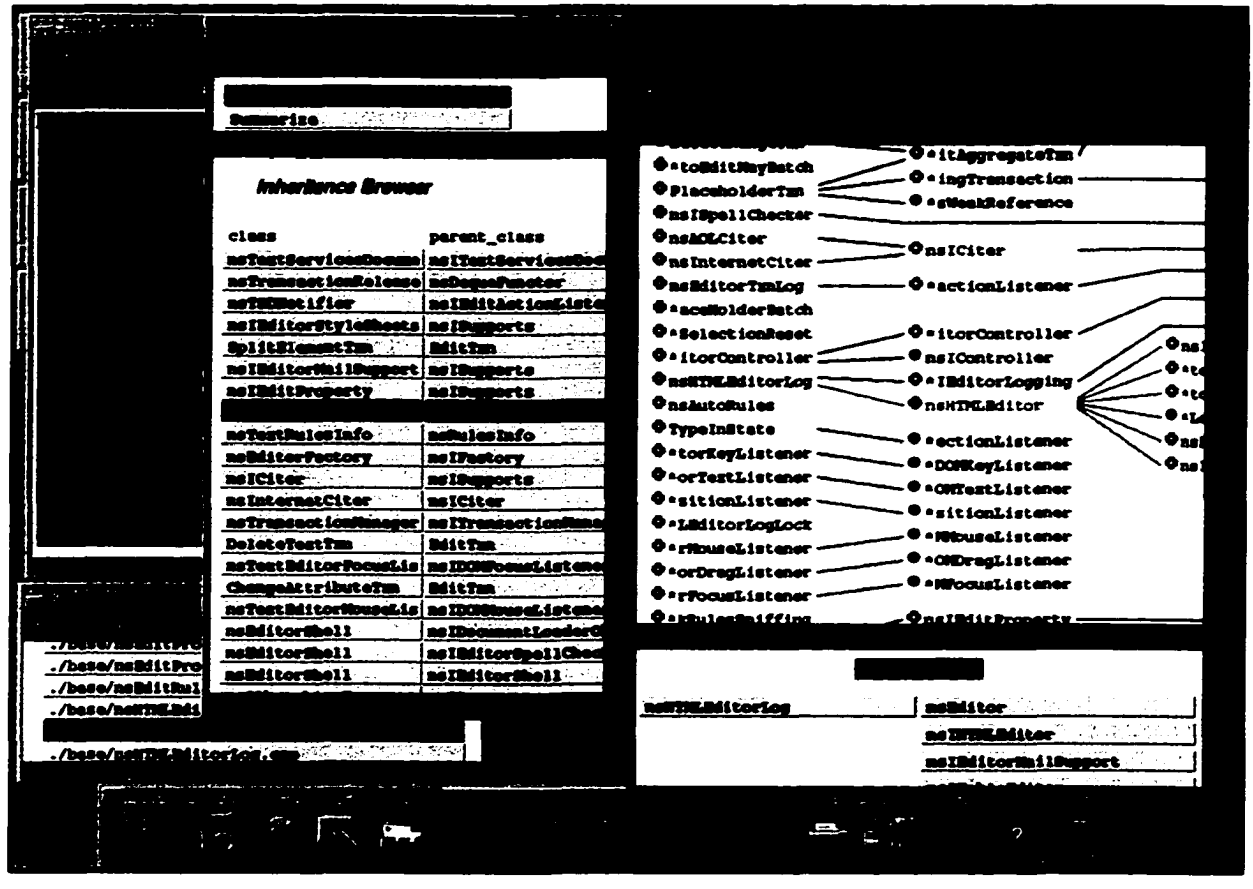

Figure 19. Audit-RE sample session

Audit-RE provides object-oriented metrics most relevant to detecting problems in legacy code and reengineering. Table 5 displays the object-oriented metrics that AUDIT-RE provides. Audit-RE also provides metrics for non object-oriented systems,

such as size, coupling, cohesion, and complexity metrics (e.g. LOC, Halstead, and McCabe's cyclomatic complexity).

Audit-RE provides problem detection or violation of best-practice heuristics in object-oriented design such as (1) detects unused components of a class, (2) identifies base classes that depend on their derived classes, (3) detects unnecessary inheritance that was used to achieve code reuse, (4) identifies multiple inheritance, and (5) detects inappropriate use of an operation as a class.

Table 5. Audit-RE object-oriented metrics

Metric	Description
Lines of Code (LOC)	Measures the complexity of a piece of source code by counting the lines.
Depth in Inheritance Tree (DIT)	Measures the depth of a class in the system's inheritance tree.
Number of Children (NOC)	Counts the number of children (direct subclasses) of a class.
Number of Methods (NOM)	Counts the number of methods in a class.
Number of Descendants (NOD)	Counts the number of descendants (direct and indirect subclasses) of a class.
Response Set for a Class (RSC)	Measures complexity and coupling properties of a class by evaluating the size of the response set of the class, i.e. how many methods (local to the class and methods from other classes) can be potentially invoked by invoking methods from the class.
Tight Class Cohesion (TCC)	Measures the cohesion of a class as the relative number of directly connected methods. (Methods are considered to be connected when they use common instance variables.)
Change Dependency Between Classes (CDBC)	Determines the potential amount of follow-up work to be done in a client class when the server class is being modified, by counting the number of methods in the client class that might need to be changed because of a change in the server class.
Data Abstraction Coupling (DAC)	Measures coupling between classes as given by the declaration of complex attributes, i.e. attributes that have another class of the system as a type.
Weighted Method Count (WMC)	Measures the complexity of a class by adding up the complexities of the methods defined in the class. The complexity measurement of a method is the McCabe cyclomatic complexity.
Reuse of Ancestors (RA)	Measures how much of a class is reused from one of its super classes.

4.5. The Integrated Reengineering Environment

We now describe the reengineering environment which realizes the methodology through the integrated set of tools.

Recall, the methodology consists of six steps, (1) reverse engineering, (2) subsystem decomposition, (3) object-based adaptation, (4) wrapping and interface

definition, (5) allocation, and (6) implementation (see Figure 2). The steps are performed in a sequential way, and the output of early steps is the input of later steps. We introduced several research and industrial software tools that provide functionality for different purposes, such as parsing, design recovery and visualization, software metrics, and data mining.

The input to the methodology is the legacy code and the output is the component-based distributed system. In addition, the set of tools that we introduced provides the support for some of the steps of the methodology. The objective was to establish an environment that contains a sets of tools that can be integrated (i.e., such tools must provide interfaces to be able to work with each other) and that could realize the methodology.

Figure 20 shows the steps of the methodology and the set of tools supporting the methodology. The left side of the figure shows the six steps of the methodology, starting with the input at the top and ending with the final result at the bottom. The right side of the figure shows a mapping of the tools supporting each of the steps of the methodology.

For the reverse engineering of object-oriented systems we use SNiff+, Tablegen, and Goose. We use the parser included in the RE-ISA tool for the reverse engineering of non object-oriented systems.

There are two main tasks to perform during the decomposition of object-oriented systems, (1) software metrics and (2) data mining. We use CodeCrawler and Audit-RE for the object-oriented metrics extraction, and we use RE-ISA and the IBM Intelligent Miner to perform the association rules mining functions. For the subsystem

decomposition, we use RE-ISA. RE-ISA integrates mining and clustering algorithms to decompose a non object-oriented legacy system into a hierarchy of data-cohesive subsystems.

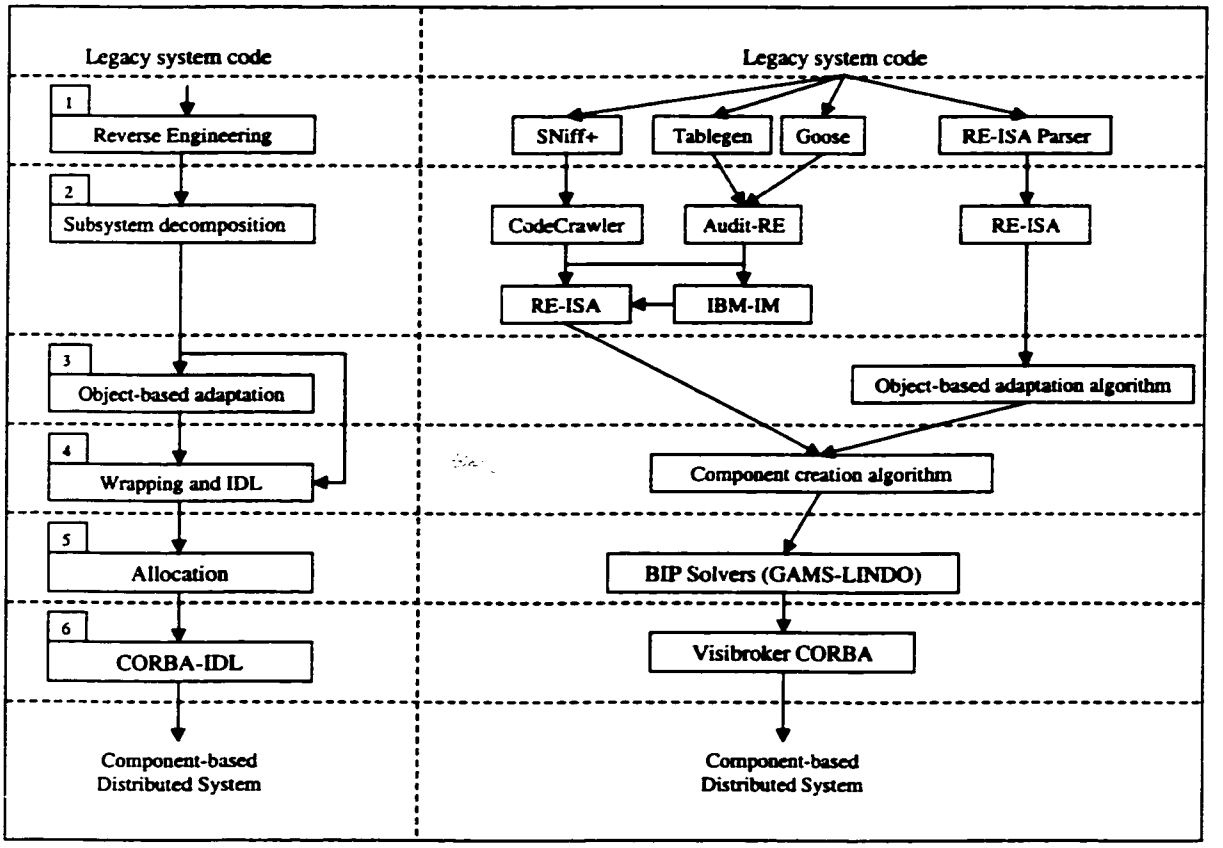

Figure 20. The reengineering environment

For the object-based adaptation we follow the algorithm that we defined in Figure 6. For the component creation we follow the rules that we defined in Section 3.4. For the allocation we use BIP solvers (i.e., GAMS, LINDO, and MS Excel) to solve the integer programming model. Finally, we use the Borland Visibroker implementation of CORBA for the implementation of the distributed system.

CHAPTER 5. CASE STUDIES

We present three case studies to show the feasibility of the methodology. The first case study is a legacy system in COBOL. The second case study is a small object-oriented program. The third case is a large object-oriented system. It is important to note that existing programs or systems represent important assets of companies, where business rules, operations and expertise are implemented. There are case studies and results reported in the literature, but the code is seldom available because of non-disclosure agreements [Deme99a]. This situation makes it difficult to do empirical studies, and even harder to publish results; consequently, empirical studies are seldom compared or reproduced.

Sections 5.1 and 5.2 present the methodology applied to the three case studies.

5.1. Case Study Using a non Object-Oriented System

We apply the methodology to a Teachers Retirement System (TRS) which is a legacy system. TRS, which runs on an IBM mainframe, consists of approximately 25,000 lines of COBOL code distributed into 28 source code files. TRS is a monolithic system without adequate documentation. For the analysis of TRS, each source code file is considered an independent *program*, and each data repository defined by a "SELECT" statement is considered a *file*. TRS has 28 programs and 38 files [Mont99], [Mont98a].

Recall, the approach for the migration of non object-oriented systems includes six steps, (1) reverse engineering, (2) subsystem decomposition, (3) object-based adaptation, (4) wrapping and interface definition, (5) allocation, and (6) implementation.

For the first step we use the integrated COBOL parser included in the RE-ISA tool. RE-ISA recovered information about the 28 programs and 38 files and their 'program-uses-file' relationships.

In the second step, we identify subsystems. We use ISA for the system decomposition. ISA organized 20 programs (71%) in 5 main subsystems. One subsystem (S3) has two primitive subsystems (i.e., S3.1, and S3.2). Only 8 programs were not assigned to any subsystem. Figure 21 shows the subsystem decomposition of the system.

From the 38 files in the system, one file (f17) is a common file, 22 files (57%) are assigned to subsystems, and 15 (39%) are unconnected files. Figure 22 shows the graphical representation of the subsystem decomposition into five subsystems (i.e., S1, S1, S3, S4, and S5). In Figure 22, the boxes represent programs and the circles are files. Links between two elements mean external file usage.

Figure 23 shows that subsystem S3 is a complex subsystem and has two primitive subsystems (i.e., S3.1 and S3.2). In Figure 23, hexagons mean links (file usage) with other subsystems.

In the third step, we create objects. First, we create six objects denoted OSS1, OSS2, OSS31, OSS32 OSS4, and OSS5, corresponding to each of the six primitive subsystems (i.e., S1, S2, S3.1, S3.2, S4, and S5). Next, we create the object OCF17 corresponding to common file f17. We create objects OCF1 and OCF2 corresponding to files f1 and f2 because they are assigned to a complex subsystem (i.e., S3). We create fifteen OUF objects corresponding to the fifteen unconnected files. These objects are denoted OUF*n*, where *n* is the number of the file. For example, OUF30 is

the object corresponding to unconnected file f30. Finally, we create four OUP objects (i.e., OUP3, OUP4, OUP11 and OUP12) corresponding to the four unconnected programs, and four OSP objects (i.e., OSP20, OSP17, OSP21 and OSP22) corresponding to the four singular programs.

```
COMMON FILES
  Files: f17

SINGULAR PROGRAMS
  Programs: p20, p17, p21, p22

UNCONNECTED ENTITIES
  Programs: p3, p4, p11, p12
  Files: f30, f22, f16, f26, f21, f28, f25, f4, f35, f27, f36, f31, f37, f29, f38

SUBSYSTEMS
  S1
    Programs: p18, p19, and p16
    Files: f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f18, f19, f20, f23, f24
        External files used: f1, f2, f17
  S2
    Programs: p23, p28, p26
    Files: f32, f33, f34,
        External files used: f1, f2, f17
  S3
    Files: f1, f2,
      S3.1
        Programs: p5, p6, p8, p13, p14, p15
        Files: f3
            File used: f1, f2
      S3.2
        Programs: p1, p2, p9, p10
            Files used: f1, f2
  S4
    Programs: p7, p24
    Files: f5
        External files used: f1, f2
  S5
    Programs: p25, p27
        External files use: f1, f3, f17
```

Figure 21. Subsystem decomposition of TRS

In the forth step, we create the components. The components are C1, C2, C31, C32, C4, and C5, one for each OSS (i.e., OSS1, OSS2, OSS31, OSS32, OSS4, and OSS5 respectively). We use the size of the component measured in number of programs as the stopping condition for the joining of components. In addition, we use

$s \leq 15$. We join C31 and C32 into one component C3. Next, we allocate all the other objects to one of the five components. OCF1 and OCF2 go to C3. OCF17 goes to C1. OSP17, OSP20, OSP21, and OSP22 go to C3. Then, OUP3, OUP4, OUP11, and OUP12 go to C3. Each OUF goes to the component having the file that accesses it.

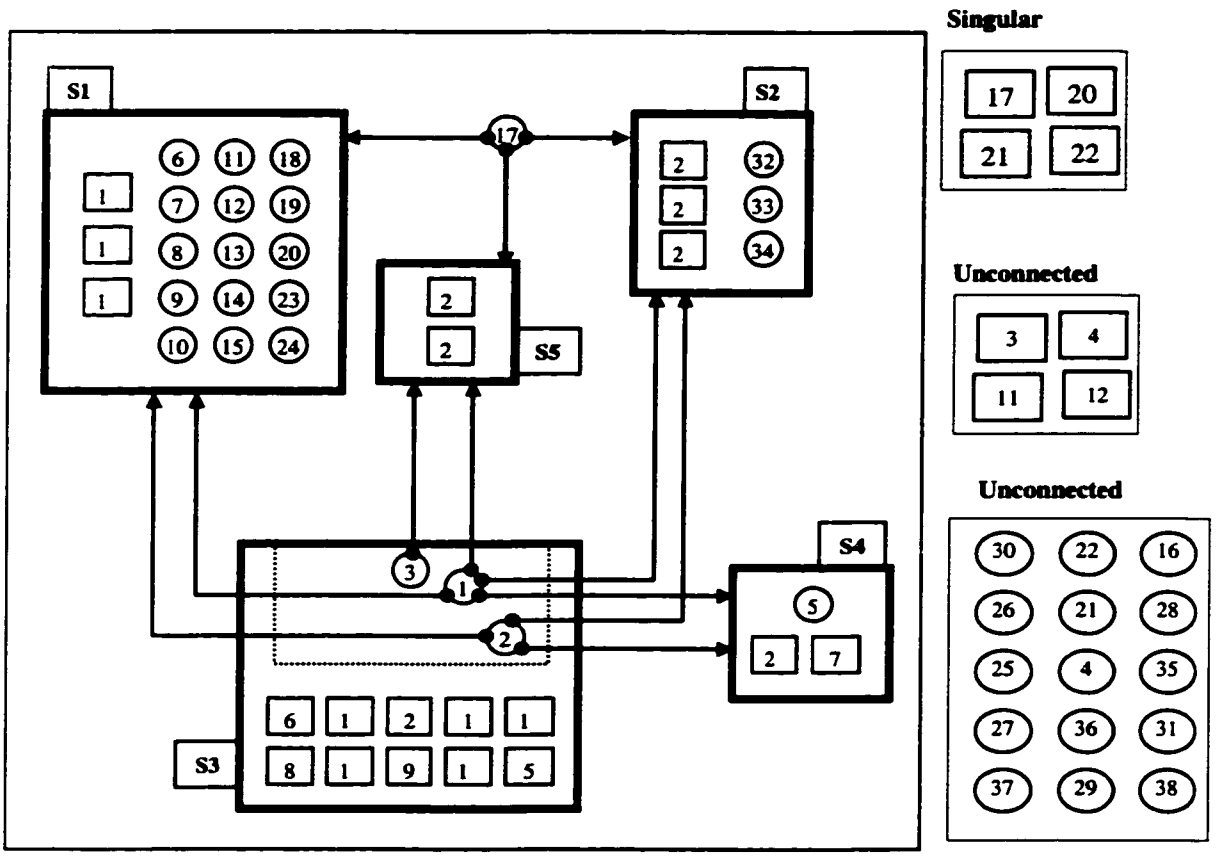

Figure 22. Graphical representation of TRS

For the fifth step, we had a LAN of PC workstations with four nodes and no hierarchy. Since the TRS system is a relatively small system, it is feasible to migrate TRS from the mainframe to the LAN. We allocate C2 and C4 to one node, because both of them are small and share several files. Finally, we allocate C1, C3, and C5, one to each remaining node in the LAN.

In the sixth step, we define the interfaces in IDL for each component. The ORB of CORBA provides the underlying communication framework that takes care of the transparent interaction between the components.

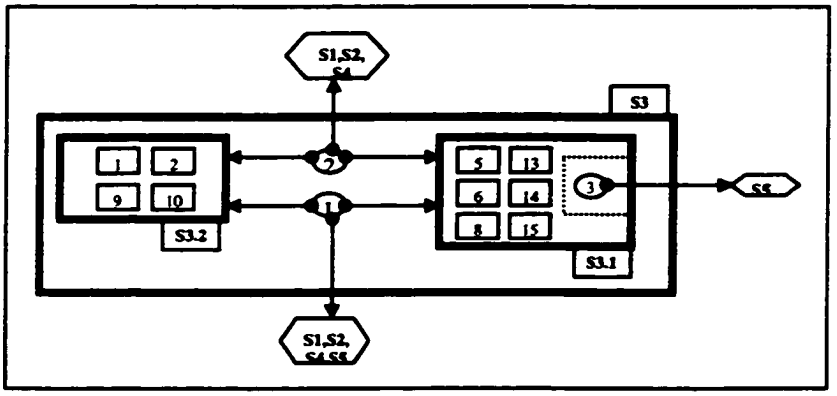

Figure 23. Graphical representation of S3

5.2. Case Study Using Object-Oriented Systems

In this section, we apply the methodology to two object-oriented systems. The first case study is a very small object-oriented system that helps us to show how the methodology works. The second case study is the C++ source code of the Mozilla-Netscape Communicator [Mozi00]. Sections 5.2.1 and 5.2.2 present the two case studies respectively.

The approach for the migration of non object-oriented systems (as stated in Chapter 3) includes several steps, (1) reverse engineering, (2) subsystem decomposition, (3) Do Nothing (4) wrapping and interface definition, (5) allocation, and (6) implementation.

5.2.1 Object-Oriented System Case Study I

We show how the methodology works when it is applied to a small object-oriented system/program. The example is small enough to be analyzed without the

need of reengineering tools. The object-oriented source code of the program is shown in Figure 24. The code consists of four classes (i.e., C1, C2, C3, and C4). Classes C3 and C4 inherit from classes C1 and C2 respectively. There are several kinds of interactions and relationships between the program constructs or entities (i.e., classes, methods, and objects).

```
class C1 {
        C2   c2o1inc1;

        public void c1m1(){
            C2   c2o1inc1m1;

            c2o1inc1m1.c2m1();
        }
        public static void c1m2Inc() { ...}
        public void c1m3(){
            c2o1inc1m1.c2m1();
        }
}

Class C2{
        private int sum;
        public c2m1(){
            c1.c1m2();
        }
        public c2m2increment(){
            sum := sum +1;
        }
}
```

```
Class C3 extends C1{
        public c3m1( C2  c2o1inc3m1par){
            c2o1inc3m1par.c2m1();
        }
        public c3m2(){
            c2  c2o1inc3m2;

            c2o1inc3m2.c2m2increment();
            c3m1(c2o1inc3m2);
            c3m1(c2o1inc1);
            }
}

Class C4 extends C2{
        C1   c1o1inc4;

        public c4m1(){
            C3    c3o1inc4m1;

            c3o1inc4m1.c3m1();
        }
        public c4m2(){
            c1o1inc4.c1m1();
            c2m1();
        }
}
```

Figure 24. Sample object-oriented code

The subsystem decomposition of this program is straightforward. We follow the steps given in the methodology. Step 1 is the reverse engineering of the system. The system is very small, it has 4 classes, (i.e., C1, C2, C3, and C4). Classes C3 and C4 inherit from classes C1 and C2 respectively.

In step 2, we generate the pairs of related classes (e.g., [C1, C2], [C1, C3], [C1, C4], etc.). Using the object-oriented metrics and metrics sets we can specify which entities have relationship with each other. Figure 25 shows the CBO and DAC metric sets from the code of Figure 24. It also shows the CBO' interaction matrix that indicates which pairs of classes are involved via CBO' coupling. Then, we make use of object-oriented metrics to generate the metrics sets. Figure 25 shows the metrics sets. The interaction matrices drive the clustering process. During the clustering, classes C1 and C3 belong to one cluster and classes C2 and C4 belong to another cluster.

Step 4 is the component creation, wrapping, and interface definition. We wrap together the objects belonging to the same cluster (e.g., C1 and C3). The interface of the component is the union of the interfaces of the respective classes.

Step 5 is the allocation of the components to nodes in a network. We can allocate the components to the same node or to different nodes. We have a LAN with four workstations. Since the components are very small, the result is that the BIP solver allocates all the components in only one site. To get different results, we specified an integer programming model with stronger constrains specifying the size of memory in nodes not large enough to host all the components. The result was the allocation of the components to two different nodes, the component containing class C1 and class C3 in one node, and the component containing classes C2 and C4 in another node.

Finally, in step 6 we define the interfaces of the components in IDL, and we use CORBA to provide the communication between the components. This step of the methodology is time consuming and requires a lot of manual work. Tools for

developing distributed object systems allow us to generate the IDL interfaces from the components almost automatically. However, the adaptation of the code to provide the CORBA implementation (e.g., initialize ORB, and create and register server objects) has to be done manually.

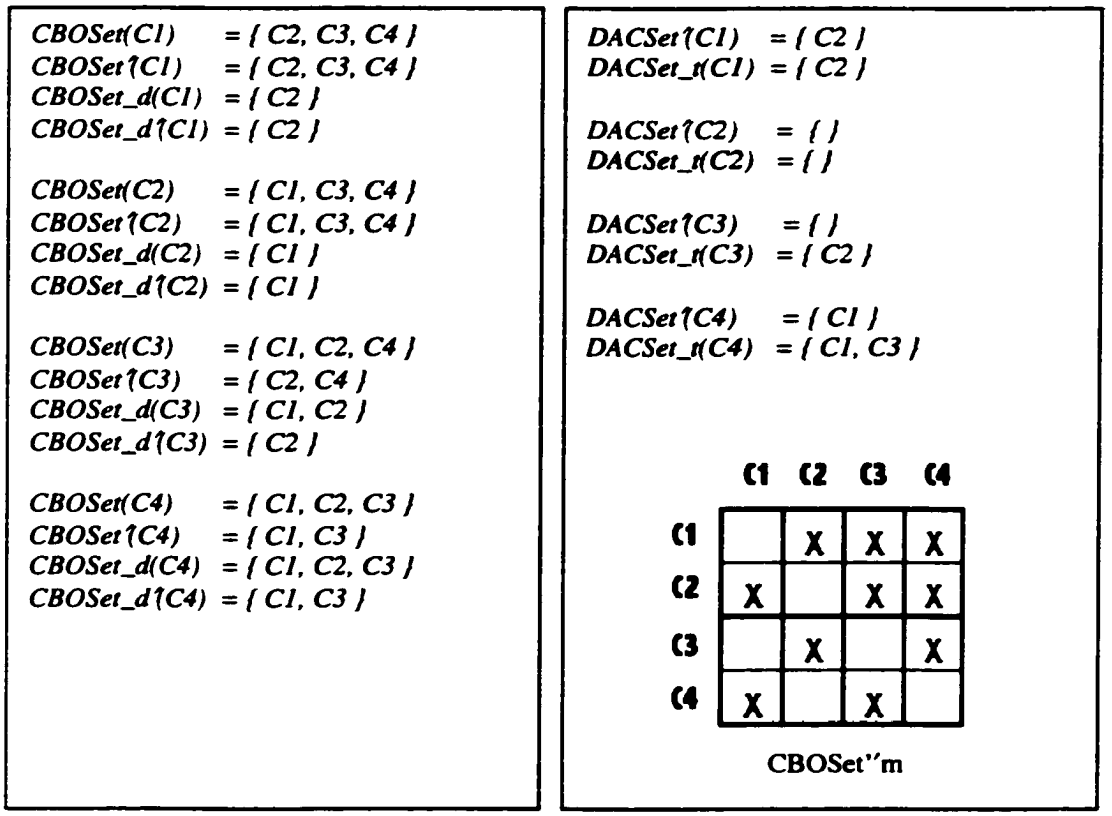

Figure 25. Metrics sets and interaction matrix

We show a simple example of an IDL interface definition and the JAVA-CORBA (server and client) implementation for the sample code. In this example we consider class C3 as the client class and class C2 as the server class. C2 provides the c2m2increment() service to class C3. Figure 26 shows the IDL definitions for the C2 server interface. The IDL interface is generated automatically from JAVA code using the Java-To-IDL translator.

```
// C2.idl
module ModuleC2
{
  interface C2
  { attribute long sum;
    long c2m2increment();
  };
};
```

Figure 26. IDL interface

The server side of count has two parts, one is the main program that initializes the ORB, creates an object of type C2 and exports that object to the ORB. The second is the implementation of the server class. Figure 27 shows the server main program.

```
// C2Server.java: The C2 Server main program

class C2
{ static public void main(String[] args)
  { try
    { // Initialize the ORB
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

      // Initialize the BOA
      org.omg.CORBA.BOA boa = orb.BOA_init();

      // Create the C2 object
      C2Impl c2 = new C2Impl("C2 object");

      // Export to the ORB the newly created object
      boa.obj_is_ready(c2);

      // Ready to service requests
      boa.impl_is_ready();
      }
      catch(org.omg.CORBA.SystemException e)
      { System.err.println(e);
      }
    }
}
```

Figure 27. CORBA server main program implementation

The implementation of the server class C2 (i.e., C2Impl) extends one of the base classes generated by the IDL2java 'compiler' (i.e., ModuleC2.C2ImplBase). Figure 28 shows the C2Impl server class implementation.

```
// C2Impl.java: The C2 Implementation
class C2Impl extends ModuleC2._C2ImplBase
{
  private int sum;

  // Constructors
  C2Impl(String name)
  { super(name);
    System.out.println("C2 Object Created");
    sum = 0;
  }

    // c2m2increment method
  public int c2m2increment()
  { sum++;
    return sum;
  }
}
```

Figure 28. C2Impl server class implementation

The client side (class C3) initializes the ORB, binds to the C2object (i.e., server object), and invokes the routine to increment the C2object. Figure 29 shows the implementation of the client side.

```
// C2Client.java  C3 class, VisiBroker for Java
class C3
{ public static void main(String args[])
  { try
    { // Initialize the ORB
      System.out.println("Initializing the ORB");
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);

      // Bind to the C2 Object
      System.out.println("Binding to C2 Object");
      ModuleC2.C2 objectC2 = ModuleC2.C2Helper.bind(orb, "C2 Object");

      // Increment C2object
        objectC2.c2m2increment();
    } catch(org.omg.CORBA.SystemException e)
    { System.err.println("System Exception");
      System.err.println(e);
    }
  }
}
```

Figure 29. CORBA client implementation

5.2.2 Object-Oriented System Case Study II

We show how the methodology works when it is applied to a large object-oriented system. The system that we consider in this section is a portion of code of the

Mozilla-Netscape Communicator, specifically, the editor/html-composer. Mozilla.org [Mozi00] created by the Netscape Corporation is a dedicated team and web site supporting development of free client source code. Mozilla is the open source code of the Netscape Communicator without all the code that Netscape is unable to release due to license or export restrictions. Mozilla is being developed by the free software community with the cooperation and support of Netscape.

Mozilla-Netscape has many components, such as the Web browser, HTML Composer/Editor, Mail/News, security/encryption, and JavaScript. It is written in C and C++, it has 1223 projects in 6713 files. Figure 30 shows some statistics of the Mozilla code.

MOZILLA

SUMMARY Symbol Table Statistics of 1223 projects

Files: 6713
Includes: 36492
Macros: 27024
Functions: 15898
Types: 3176
Variables: 11151
Enums: 715
Userdef: 0
Classes: 5933
  Instance Variables: 23757
  Methods: 41015
  Friends: 273
  Localdefs: 290

SUMMARY File Type Statistics of 1223 projects

File Type	Number of files
HTML	763
Header	3331
Implementation	3382
Make	117
Project Description	1309

Figure 30. Statistics of Mozilla code

Mozilla has a very complex code and design. Figure 31 shows a SNiff+ screen shot of a portion of the code showing the inheritance tree (graph) and the class declaration of a class with multiple inheritance.

In this section, we apply our methodology to the HTML Composer/Editor component of the Mozilla code.

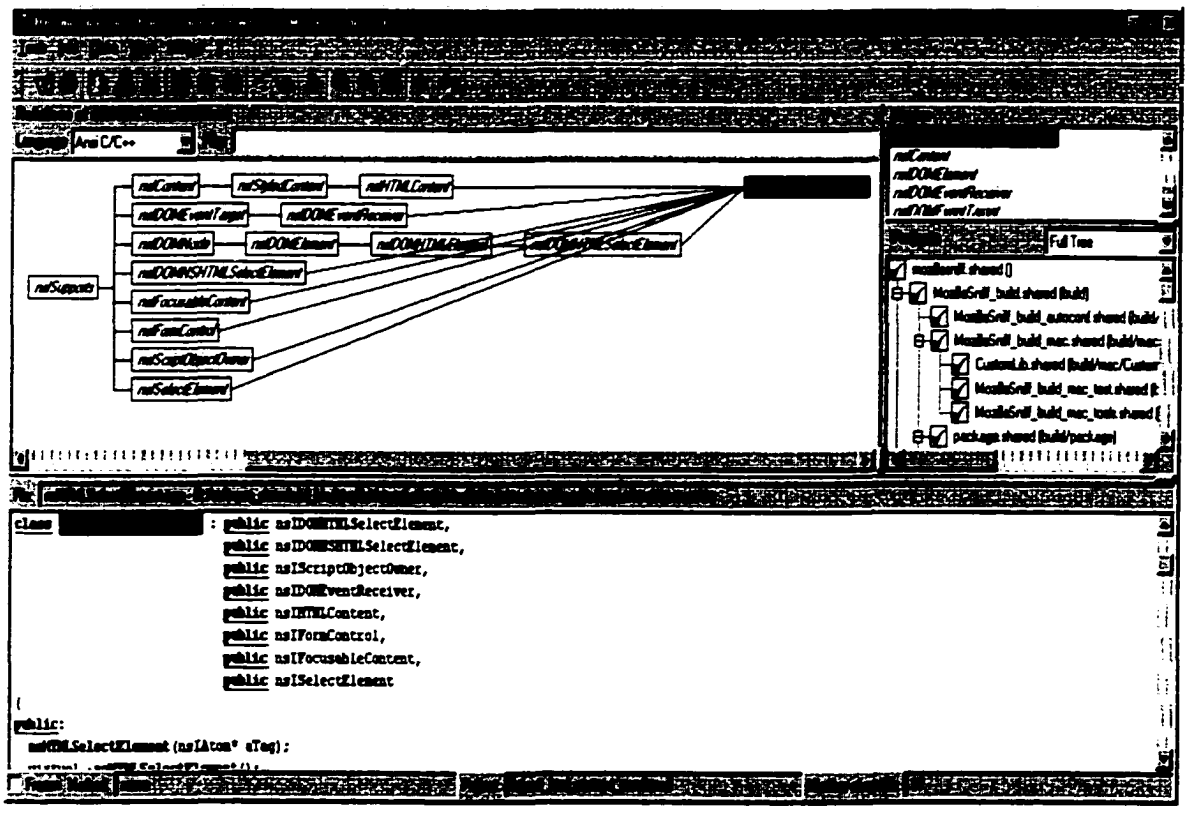

Figure 31. Excerpt of Mozilla hierarchy tree

The first step of the methodology is the reverse engineering and design recovery of the code. This step is of most importance, not just for the information gathering needed for the methodology, but for the team performing the migration. Before any reengineering activity takes place, understanding of the legacy system architecture and design by the team is necessary. Comprehension of the different entities and

relationships present in the legacy system are necessary for any successful reengineering and migration activity.

This section may give the reader the impression that too many tools are used and too many screens are captured and shown here. We believe that these few screens can not give a real idea of the complexity of the system and the reengineering tasks. Only real experimentation with the tools will do it, but we try to show some views and screen shots from the different tools.

We use SNiff++ to parse the code and retrieve information needed for the following phases. Figure 32 shows a SNiff++ screen shot with a portion of the inheritance tree (graph) of the HTML-editor code. The inheritance tree gives us a quick idea of how the code is organized. In the graph, the nodes are classes and the edges are inheritance relationships. SNiff++ allow us to navigate the graph and the actual code.

SNiff++ also gives us general information about the system. The HTML Editor has 30 projects in 111 files, it has 90 classes with 415 instance variables and 1320 methods. Figure 33 shows some other global statistics of the HTML Editor.

We export the parsing information from SNiff++ into the FAMIX CDIF format, and analyze the code using CodeCrawler. Figure 34 shows us the inheritance graph (system complexity graph) of the HTML Editor. Navigating the code with CodeCrawler gives us further information in a friendly visual way. The metrics considered in the system complexity graphs are NIV, NOM, and WNI, corresponding to the width, height, and color of the node respectively. We can notice that this specific graph is very similar to the inheritance graph that SNiff++ gave us. The big

difference is that CodeCrawler gives us information about object-oriented metrics and shows it in a graphical manner. There are many other useful views that CodeCrawler offers us, but we do not explain all of them here because of space constraints.

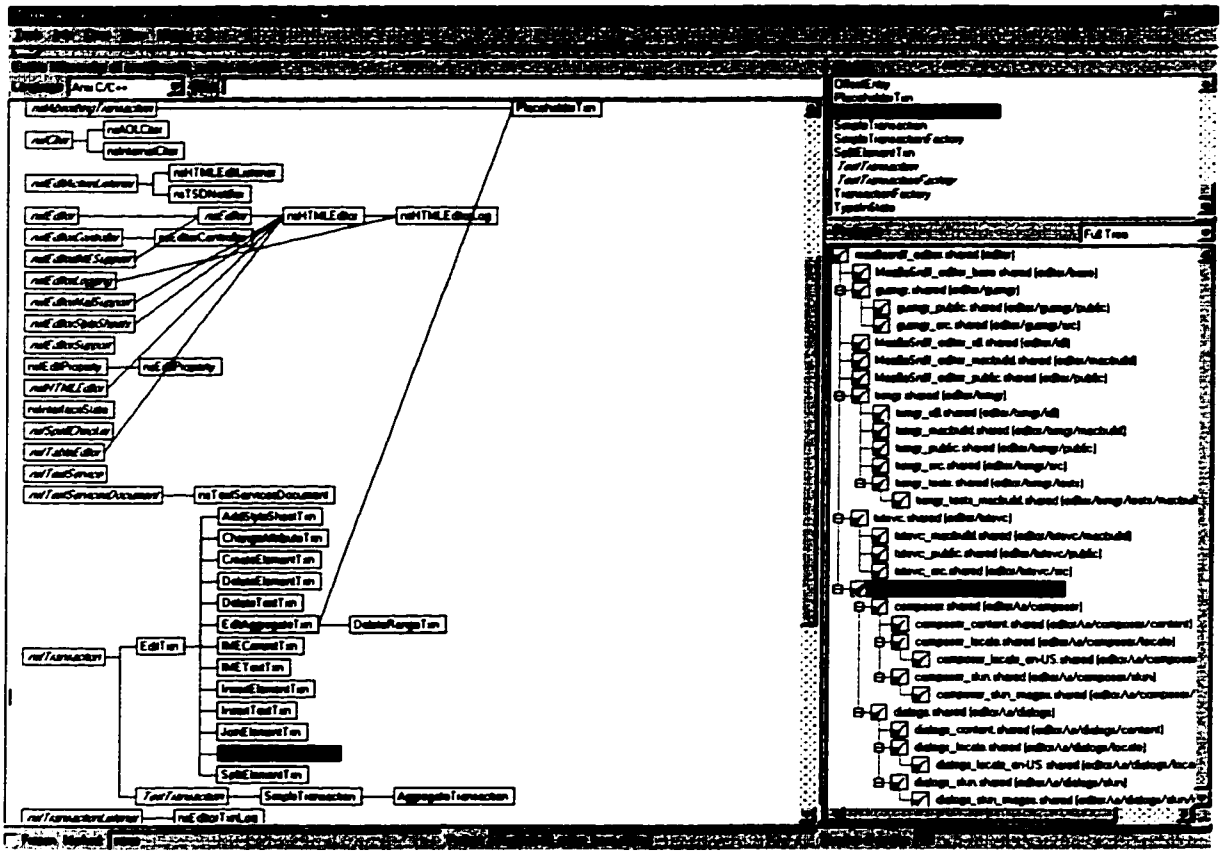

Figure 32. HTML Editor hierarchical tree

We finish the reverse engineering and design recovery phase analyzing the legacy code with one more reengineering tool, the Audit-RE.

The parsers used by Audit (i.e., Tablegen and Audit Fast Library) perform the analysis of the object-oriented code and store the information of the recovered design (i.e., object-oriented entities and relationships) in tables. Then, Audit-RE uses that information to produce different graphical views of the system, assess the quality of the design, discover violations of good design practices, and calculate object-oriented

metrics. We are more interested in the design recovery and understanding of the system (graphical views, object-oriented relationships, and metrics) than in the quality assessment of the design.

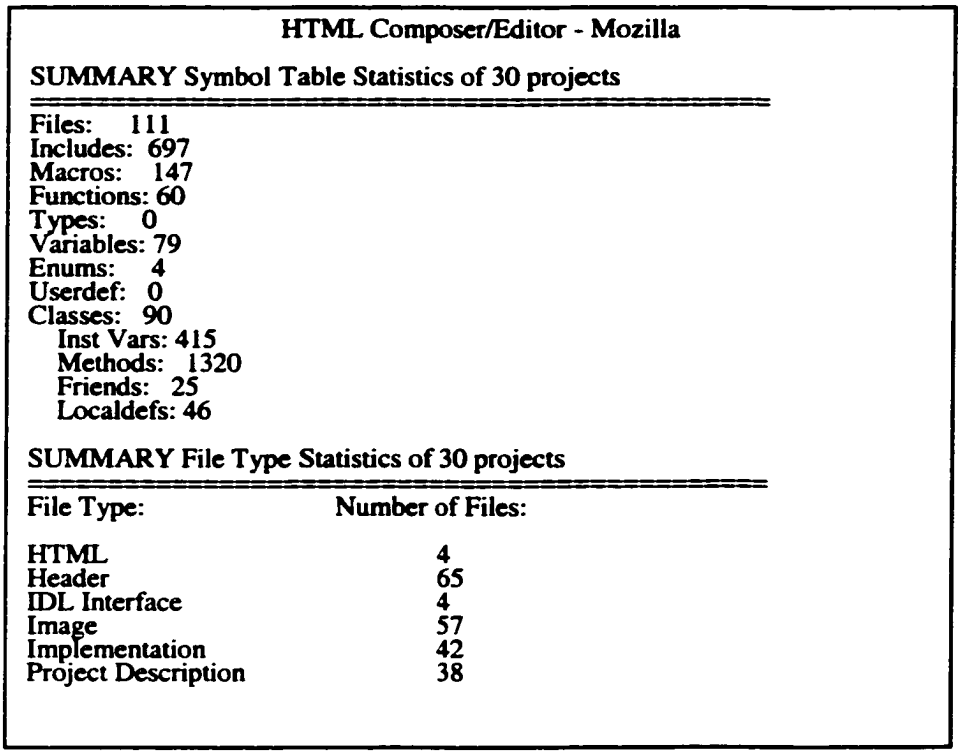

Figure 33. Statistics of HTML Editor

Figure 35 shows the inheritance graph of the application. In the inheritance graph, the nodes are classes, and if two nodes have a link, the node on the left is the child class and the node on the right is the father class.

Audit-RE also provides us with other useful graphical views. Figure 35. Audit - HTML Editor inheritance graph

shows a class-use graph. This graph shows the inheritance and DAC (Data Abstraction Coupling) relationships together. Inheritance and DAC relationships are in the following categories [Sema98]:

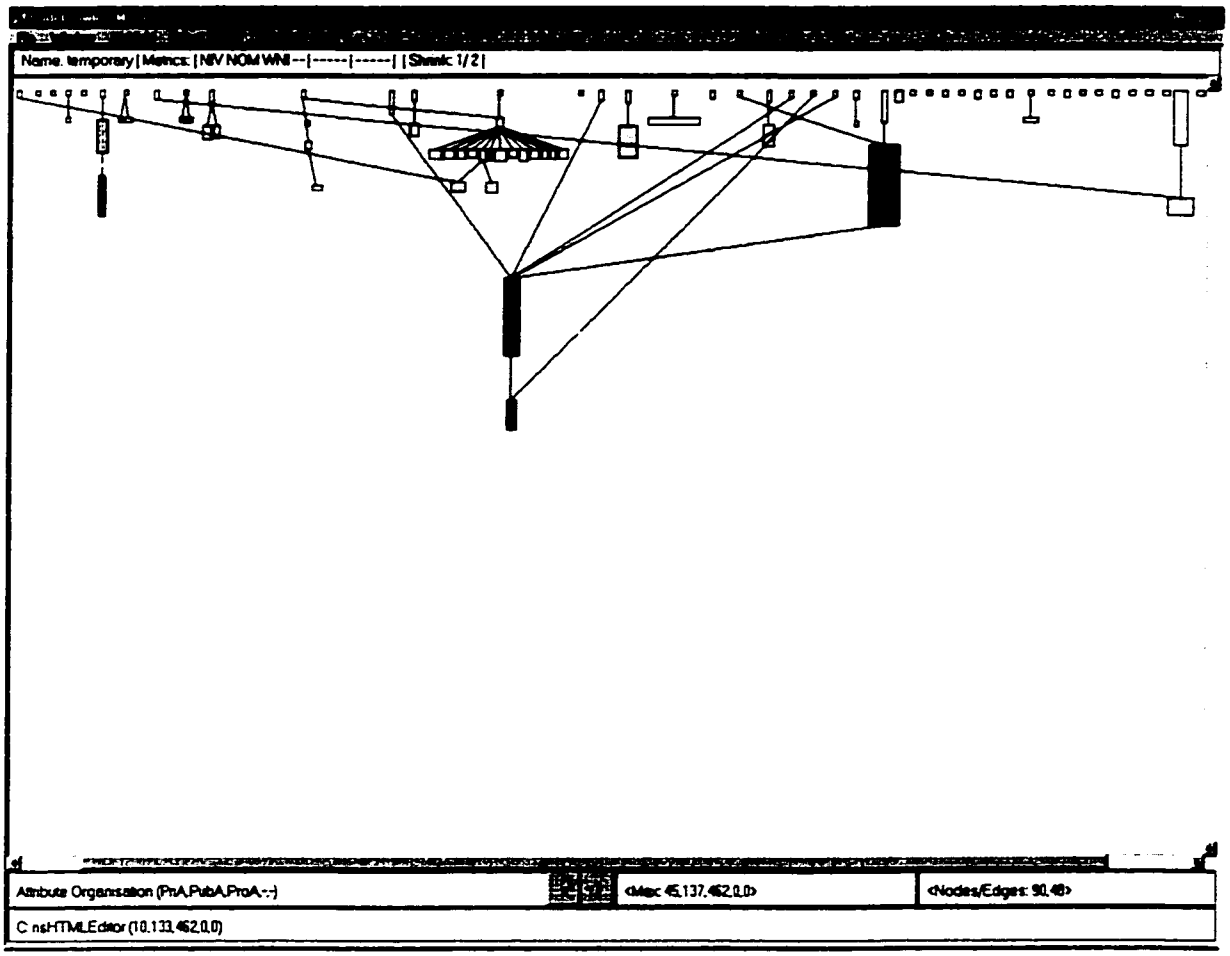

Figure 34. CodeCrawler HTML Editor system complexity view

- Characteristic use: when a C1 class derives from a C2 class (inheritance), or when C2 is used for typing a C1 member.
- Contextual use: when C2 is used for typing parameters of a C1 function member.

- Operational use: when C2 is locally used in a C1 function member body, for typing a local variable, for creating an object (by the way of the new instruction), for type conversion (cast), or for accessing a C1 static member directly.

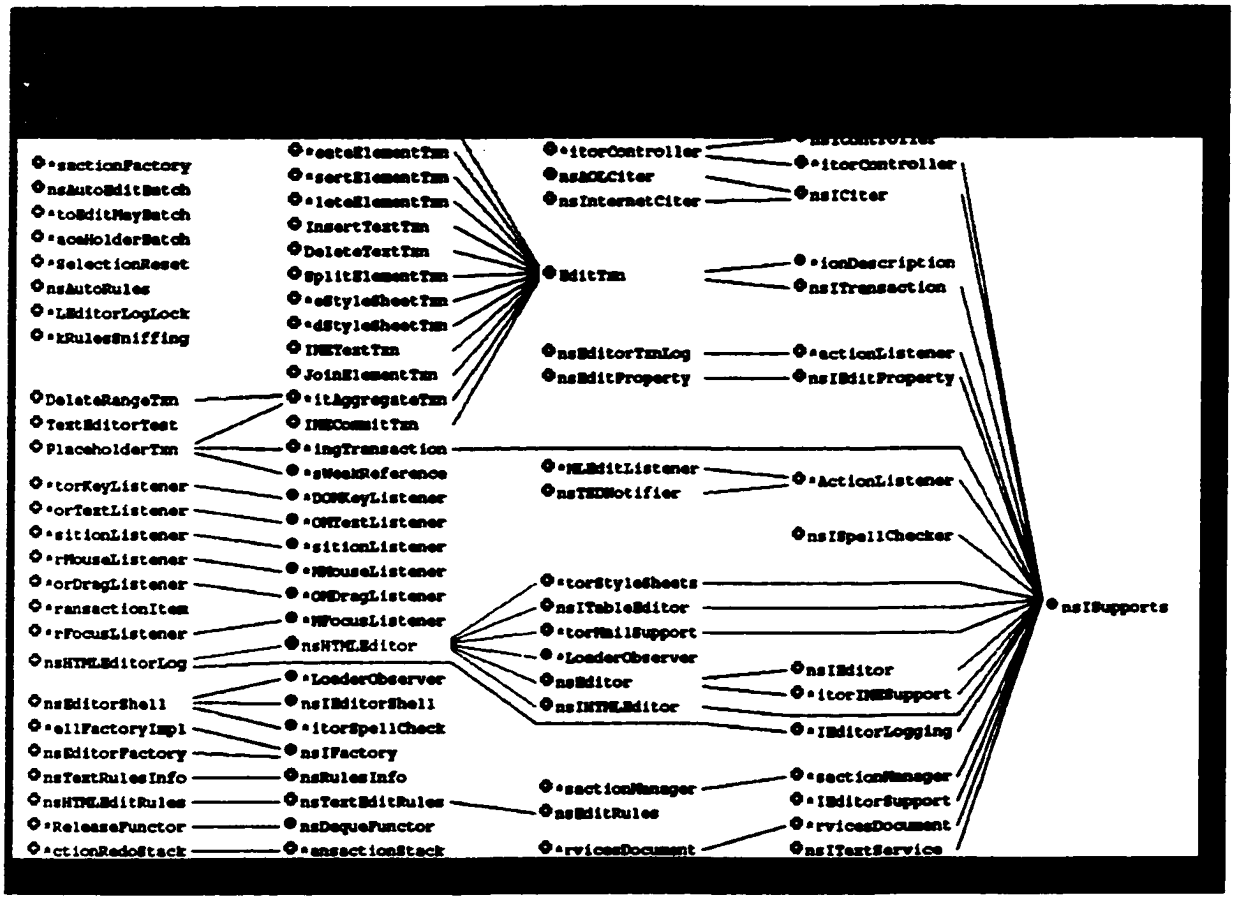

Figure 35. Audit - HTML Editor inheritance graph

The following step is the creation of the interaction matrices corresponding to the CBO, RFC, and DAC object-oriented metrics. The interaction matrices are derived from the corresponding metric sets that are generated by the tools. In order to handle the interaction matrices more easily, we map the class names to numbers. Figure 36. Audit - class-use graph

shows the name of the classes and their respective numbers.

One portion of the CBO interaction matrix is shown in Figure 38. The left most column and the first row contain the class number. A mark (i.e., X) means that the class corresponding to the row 'uses' the class corresponding to the column. The term 'uses' denotes interactions or relationships between classes (e.g., inheritance, call, and access). As an example, we say that class 5 is CBO-coupled with classes 4,6,7, and 74.

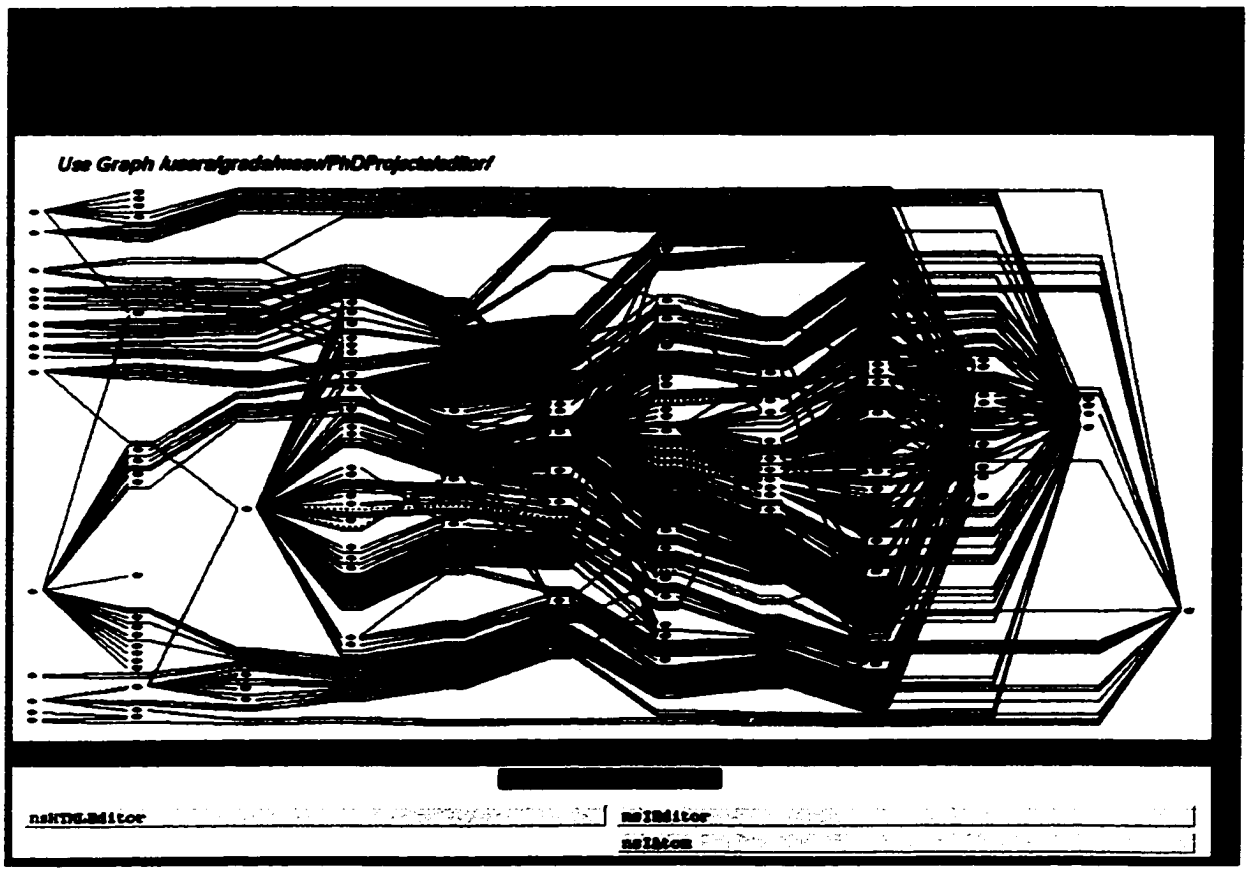

Figure 36. Audit - class-use graph

We also generate other interaction matrices corresponding to other object-oriented metrics such as RFC and DAC. It is also possible to create interaction matrices with a combination of different object-oriented metric sets. Notice that the interaction matrix

shown in the figure corresponds to the CBO_d object-oriented metric that we defined and not the most common CBO metric.

The next step is to apply data mining association-rules to the interaction matrix. We use a low value of minimum support to allow transactions (using classes) with few common items (used classes) to be considered in the result. The data mining process discovered 59 transactions, 64 items, 21 the maximum number items per transaction (corresponding to class classes 22 in the interaction matrix), and 656 association rules of length 2.

1	AddStyleSheetTxn	47	nsISpellChecker
2	ChangeAttributeTxn	48	nsITableEditor
3	CreateElementTxn	49	nsITextService
4	DeleteElementTxn	50	nsITextServicesDocument
5	DeleteRangeTxn	51	nsITransaction
6	DeleteTextTxn	52	nsITransactionListener
7	EditAggregateTxn	53	nsITransactionManager
8	EditTxn	54	nsRulesInfo
9	IMECommitTxn	55	nsTextEditorCompositionListener
10	IMETextTxn	56	nsTextEditorDragListener
11	InsertElementTxn	57	nsTextEditorFocusListener
12	InsertTextTxn	58	nsTextEditorKeyListener
13	JoinElementTxn	59	nsTextEditorMouseListener
14	nsAOLCiter	60	nsTextEditorTextListener
15	nsAutoEditBatch	61	nsTextEditRules
16	nsAutoEditMayBatch	62	nsTextRulesInfo
17	nsAutoHTMLEditorLogLock	63	nsTextServicesDocument
18	nsAutoLockRulesSniffing	64	nsTransactionItem
19	nsAutoPlaceHolderBatch	65	nsTransactionManager
20	nsAutoRules	66	nsTransactionRedoStack
21	nsAutoSelectionReset	67	nsTransactionReleaseFunctor
22	nsEditor	68	nsTransactionStack
23	nsEditorController	69	nsTSDNotifier
24	nsEditorFactory	70	PlaceholderTxn
25	nsEditorShell	71	RemoveStyleSheetTxn
26	nsEditorShellFactoryImpl	72	SplitElementTxn
27	nsEditorTxnLog	73	TextEditorTest
28	nsEditProperty	74	TransactionFactory
29	nsEditRules	75	TypeInState
30	nsHTMLEditListener	76	nsITransactionDescription
31	nsHTMLEditor	77	nsIController
32	nsHTMLEditorLog	78	nsIFactory
33	nsHTMLEditRules	79	nsIEditorSpellCheck
34	nsIAbsorbingTransaction	80	nsIEditorShell
35	nsICiter	81	nsIDocumentLoaderObserver
36	nsIEditActionListener	82	nsICSSLoaderObserver
37	nsIEditor	83	nsISupports
38	nsIEditorController	84	nsIDOMCompositionListener
39	nsIEditorIMESupport	85	nsIDOMDragListener
40	nsIEditorLogging	86	nsIDOMFocusListener
41	nsIEditorMailSupport	87	nsIDOMKeyListener
42	nsIEditorStyleSheets	88	nsIDOMMouseListener
43	nsIEditorSupport	89	nsIDOMTextListener
44	nsIEditProperty	90	nsDequeFunctor
45	nsIHTMLEditor	91	nsSupportsWeakReference
46	nsInternetCiter	92	nsIDOMSelectionListener

Figure 37. Number mapping of class names

Figure 39 shows some association rules resulting from the data mining. The figure presents the support and confidence of the association rule. In the sample shown, the

second row can be interpreted as "when a class is used by class TransactionFactory then the class is also used by class nsEditor in 100% of the cases; the pattern is present in 23.72% of transactions". We concatenated the number of the class at the end of the class name to make the analysis easier.

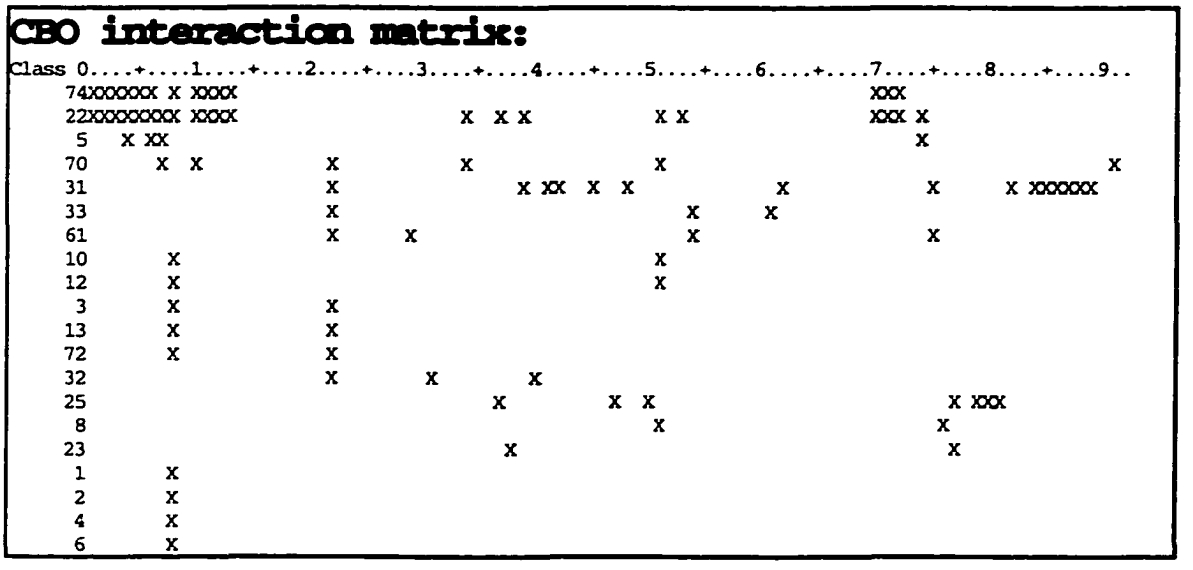

Figure 38. CBO interaction matrix of HTML Editor

Now we start the hierarchical clustering driven by the association rules. We are using 'support' as the similarity metric (association coefficients) for the hierarchical clustering algorithm. We experimented using Jaccard's coefficients and discovered that they produce similar results. Figure 40 shows the result of the clustering phase.

Hierarchical clustering produced a hierarchy with four clusters (i.e., S1, S2, S3 and S4), and many small clusters that we joined into one cluster (i.e., S5). S2 has two sub-subsystems (i.e., S2.1 and S2.2). Classes in S2.1 are the most similar (i.e., more related, or the support of their associations rules is the highest). There are some classes in S2 that do not belong to S2.1 or S2.2, they are 'common' classes to both. In S2, the

group of files inside the dotted box corresponds to class 8 and most of the classes that inherit from it. In the diagram, boxes grouping a set of classes indicate that inside the box, the similarity between classes is higher than the similarity with classes outside the box.

Support	Confidence	Body	==>	Head
23.72	66.7	[nsEditor22]	==>	[TransactionFactory74]
23.72	100.0	[TransactionFactory74]	==>	[nsEditor22]
6.780	19.1	[nsEditor22]	==>	[DeleteRangeTxn5]
6.780	66.7	[PlaceholderTxn70]	==>	[nsEditor22]
6.780	100.0	[DeleteRangeTxn5]	==>	[nsEditor22]
6.780	19.1	[nsEditor22]	==>	[PlaceholderTxn70]
3.390	100.0	[InsertTextTxn12]	==>	[IMETextTxn10]
3.390	100.0	[IMETextTxn10]	==>	[nsEditor22]
3.390	100.0	[CreateElementTxn3]	==>	[SplitElementTxn72]
3.390	66.7	[nsHTMLEditRules33]	==>	[nsTextEditRules61]
3.390	9.5	[nsEditor22]	==>	[InsertTextTxn12]
3.390	100.0	[SplitElementTxn72]	==>	[CreateElementTxn3]
3.390	14.3	[TransactionFactory74]	==>	[DeleteRangeTxn5]
3.390	100.0	[JoinElementTxn13]	==>	[SplitElementTxn72]
3.390	50.0	[nsTextEditRules61]	==>	[nsHTMLEditRules33]

Figure 39. Association rules HTML Editor

In S5 there are 4 groups (one in each doted box). In each group, the classes are related, except in the lower-left group (i.e., the group containing classes 15 and 16) in which the classes are not related. In S5, the similarity (support in the association) between classes was the lowest one.

Previously, when explaining Figure 40, we mentioned that inside a grouping box, the similarity between classes is higher than the similarity with classes outside the box. Figure 41 shows and example (with only a few classes that belong to S2) of the hierarchical three of the clustering decomposition. Classes 74 and 22 are the ones more similar. Classes 10 and 12 are less similar.

The following step is the creation of components. We create components by wrapping each subsystem and defining its interface in IDL. The interface of the component consists of the union of interfaces of the classes that belong to the

component. The interface of a class consists of the signatures of methods that provide services to other classes (objects). In the case study, only a few classes needed to be included in the IDL interface (e.g., EditTxn, nsISupports, nsEditor, and nsIEditor).

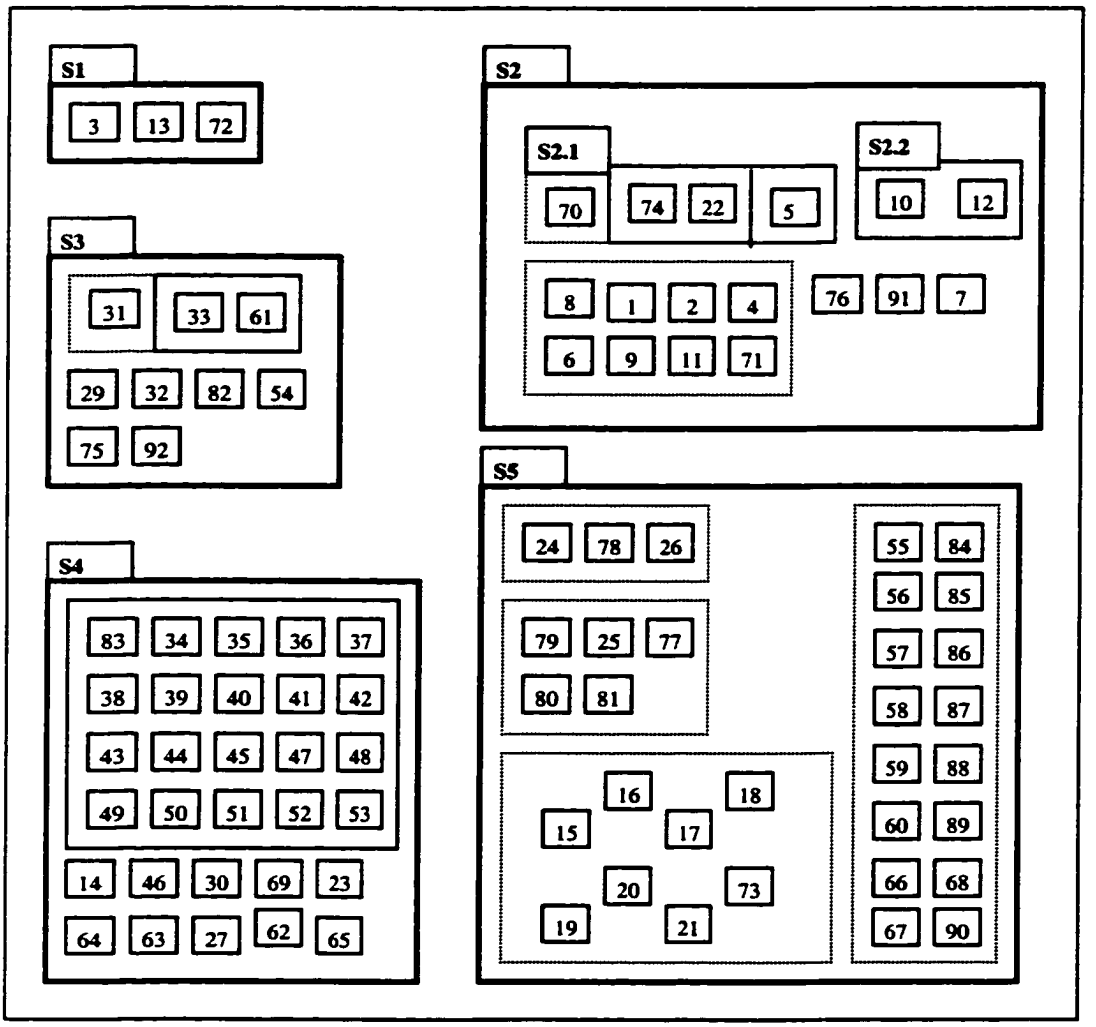

Figure 40. HTML Editor subsystem decomposition

Mozilla (Netscape) is an application that has a very complex design, with many classes and relationships between them. However, Mozilla does not process large amounts of information (numerical and data) and the requirements for memory (storage size) are not significant. The result is that the BIP solver allocates all the components in only one site. To get different results, we specified an integer

programming model with stronger constrains specifying the size of available memory in nodes not large enough to host all the components. The result was the allocation of the components to two different sites, S1, S2, and S4 in one node, and S3 and S5 in another node. Finally, we used CORBA for the communication among components in the network. Writing the CORBA-IDL interfaces as well as the code necessary for the ORB is time consuming and not automatic.

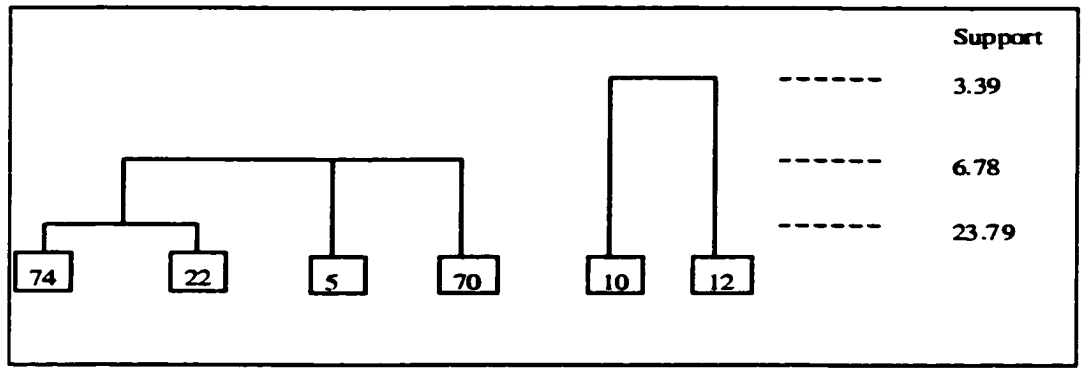

Figure 41. Tree representation of hierarchical clustering

5.3. Analysis of Case Studies

In each of these case studies, we produce distributed systems consisting of high-cohesive and low-coupled components. The systems by definition have low inter-component communications and high intra-component interactions. Low inter-component communications and high intra-component interactions are important features in any good distributed system. The case studies demonstrate that the methodology produces distributed systems with these desirable features using the information provided in the legacy code.

The methodology migrates legacy systems to distributed environments with information available only in the legacy system code, as opposed to most of the

approaches to develop distributed systems that start with new specifications and designs. The case studies demonstrate that the methodology produces an efficient decomposition, allocation, and implementation of the legacy system in a distributed environment with availability of the code only.

The methodology groups together sets of highly related and dependent entities, i.e., groups together entities that have a high coupling and high dependence into the same cluster and entities with low coupling and low dependence into different clusters. Each cluster becomes a subsystem of the original system. When we generate the components from these subsystems and then allocate them in a distributed environment, they result in a system with low coupling. Low coupling and dependences translate into low inter-site communication in the distributed system.

An advantage of the resulting distributed system is that the components can be reused and reengineered separately. In addition, by using the CORBA-IDL interfaces, elements of the system can be used or use existing functionality in other systems that are CORBA compliant. In addition, the evolutionary migration nature of the methodology is suitable for the incremental migration of legacy systems to new technologies. Once a subsystem is identified, wrapped, and converted to a component with its respective interface, it is possible to start the incremental migration of each component independently.

One of the most important features of the methodology is that it has a high degree of automation. For the first phase, there are many reverse engineering tools available that deal with the design recovery of object and non object based legacy systems. The subsystem decomposition phase can also be automated. The object-based adaptation,

allocation, and implementation of the distributed system can be automated partially; however, interaction of the user is also needed. Another unique feature of the methodology is that it does not rely on a specific implementation of data mining, clustering, and allocation algorithms. Therefore, if newer and more efficient algorithms are developed in those specific fields, we can adapt them to work with our methodology.

CHAPTER 6. SUMMARY AND CONCLUSIONS

This research was undertaken to define a feasible, semiautomatic approach for the reengineering and evolutionary migration of legacy systems to distributed environments. We defined a methodology that uses a composition of concepts and techniques such as reverse engineering, software metrics, data mining, and clustering to achieve the reengineering of legacy systems to object-based distributed environments.

Sections 6.1 to 6.4 address the summary, conclusions, contributions, and future research respectively.

6.1. Summary

Reengineering and migration of legacy systems to distributed systems is often done in ad-hoc manner. By building on research in multiple areas such as reverse engineering, data mining, software metrics, clustering, and integer programming, we have defined a methodology for the systematic migration of legacy systems to distributed environments. We also defined a reengineering environment that implements the methodology with an integrated set of tools.

The methodology presented in this research consists of the following steps:

(1) Reverse engineering techniques are used for the architectural and design recovery.

(2) A hierarchical data-cohesive decomposition of the system into appropriate distributable units is achieved by defining relationships in the underlying paradigm of the legacy system. Object-oriented metrics are used if the underlying paradigm is object-oriented; program-uses-file relationships and data mining are used otherwise.

(3) If the underlying paradigm of the legacy system is not object-based, then, object-based adaptations are performed on the subsystems.

(4) Wrapping and interface definition are performed for the component creation

(5) Allocation of components to different sites is achieved by specifying the requirements of the system and characteristics of the network as an integer-programming model. Optimization techniques (i.e., integer programming algorithms) are used to drive the allocation of components to different sites in the network, thereby, minimizing inter-site communication.

(6) Middleware technologies are used for implementation of the distributed object system.

We have divided this research in four major sections that correspond to Chapters 2 through Chapter 5. Chapter 2 presented background information. It included definitions and concepts of the different aspects of this research such as reengineering, software metrics, data mining, and clustering. Chapter 2 also presented previous work related to this research.

Chapter 3 described the reengineering approach for the migration of the legacy system to the distributed environment. Chapter 3 provided the detailed description of how the methodology works, the input, and the output. Chapter 3 illustrated in detail the different phases of the methodology that include reverse engineering of the legacy system code, subsystem decomposition, object-based adaptation, wrapping and interface definition, allocation, and use of the middleware for the implementation of the distributed system.

Chapter 4 defined an integrated reengineering environment that consists of the methodology and a tool set that supports the methodology. The tools were used to automate the implementation of the methodology. Tools for reverse engineering, visualization, refactoring, data mining, software metrics, and clustering were introduced and integrated in the reengineering environment to achieve the tasks of reengineering and migration.

Finally, Chapter 5 presented examples of the methodology applied to legacy systems. Analysis of prototype systems and case studies was performed using the reengineering environment presented in Chapter 4. The reengineering environment is critical because it is necessary for the reengineering and migration of any system consisting of more than a few hundred lines of code.

6.2. Conclusions

In this research, we achieved the migration of legacy systems to distributed environments in a systematic way with the support of a reengineering environment consisting of a methodology supported by a set of tools.

We defined a methodology for migration of legacy systems to distributed environments. This research and case studies demonstrated that the methodology is feasible for the migration of legacy systems to the distributed environment.

The methodology integrates and extends techniques and approaches from other research works to achieve the goal of migration. The adoption, adaptation, and extension of those techniques and approaches produced some other interesting results that we now describe.

Object-oriented metrics and metric sets are valuable tools to drive the creation of the database view of an object-oriented system. They can be used for specifying relationships, dependences, and interactions in object-oriented systems. The benefit of using object-oriented metrics as opposed to just using information from the design recovery is that object-oriented metrics take into account features such as inheritance, polymorphism, and transitive relationships.

We introduced the concept of interaction matrices. The concept of an interaction matrix as a mathematical and visual representation of the metric sets was shown to be valuable for driving the data mining algorithms. In the interaction matrix, each row represents a transaction and the columns represent the attributes, items or transaction description.

The CBO_d, CBO_d', and DAC_t coupling metrics that we defined, as well as the corresponding metrics sets CBOSet_d, CBOSet_d', and DACSet_t were shown to be useful object-oriented measures to drive the decomposition of subsystems. The traditional CBO and DAC object-oriented metrics do not distinguish between interactions A $\rightarrow$ B and B $\rightarrow$ A (i.e., they are undirected). The 'directedness' feature of the metrics that we proposed produces a more accurate decomposition because the metrics identify which class depends (is coupled with) on which class. In traditional CBO and DAC metrics when two classes A and B are coupled (A $\leftrightarrow$ B) there is no distinction if A depends on B, B depends on A, or both.

Data mining produced meaningful information from the legacy system without previous knowledge of the domain. In addition, data mining algorithms were able to process large amounts of information for the analysis of large legacy systems.

Previous research on the application of data mining to non object-oriented legacy systems proved these facts. This research extended to object-oriented legacy systems.

The use of support and confidence from the data mining association rules as similarity metrics (i.e., association coefficients) to drive the clustering algorithms is a novel approach that produces a decomposition of subsystems with high-coupling. Previous research using clustering for decomposition of software systems uses other association coefficients such as Jaccard's coefficients, simple matching, and Sorencsen-Dice. We used Jaccard's associations coefficients and support associations coefficients. The results were similar, but support association coefficients produced smaller subsystems with lower coupling.

The algorithm used for the object-based adaptation of subsystems was demonstrated to produce objects that contain data-cohesive methods and attributes (see Theorem 1). In addition, the algorithm used to create components from non object-oriented legacy systems was demonstrated to produce components that contain data and processing cohesive objects (see Theorem 2). The components with these features are highly suitable for distribution.

Reengineering and migration tasks must rely on a methodology, and they must be supported by a set of tools. The reengineering environment that we defined provides both, the methodology and tool support. The task of reengineering must not be seen as a simple task that can be achieved without integrated support; but rather, it must be considered as a significantly complex and time consuming task that needs tools such as tools for reverse engineering, design recovery, visualization, and software metrics.

The reengineering environment not only allows migration of legacy systems to distributed environments, but it also produces other valuable results. The reverse engineering, software metrics, and decomposition of the legacy system provide an analysis and understanding of the legacy code without knowledge of the system beyond the code. This capability is very useful, given that many legacy systems lack documentation.

One of the limitations of our methodology is that some steps cannot be fully automated (e.g., object-based adaptation, and CORBA code generation). The lack of total automation is a major disadvantage for the migration of large legacy systems to distributed environments, since a significant amount of manual work has to be performed for complex systems. However, it is a feasible approach when compared to the total redevelopment of the system without a systematic, integrated methodology and associated environment.

6.3. Contributions

The clear advantage of object-based distributed systems makes the migration of legacy systems to new technologies not just an attractive choice but rather a necessity for many organizations that are under economical pressure by the competition. In the last years, technologies such as the internet and e-commerce have precipitated the reengineering of standalone legacy systems to client/server and distributed systems.

The main contribution of this research is that it provided an integrated reengineering environment with a methodology to migrate legacy systems to distributed environments, supported by a set of tools.

A major contribution of this research is that it provides a methodology that facilitates the systematic migration of legacy systems to distributed environments by defining a reengineering environment that supports the practical implementation and automation of the methodology. Migration and reengineering of legacy systems is a time-consuming task. To reduce the cost of these tasks, some degree of automated support is desirable.

Another major contribution of this research is that the methodology produces a distributed system that can be integrated with other applications and COTS (Components Off The Shelf) components through CORBA. Portions of the original system now in the form of components can be used/reused from other applications with the appropriate call to the interface of the component using IDL. In addition, the resulting distributed system can use/reuse functionality from any application that is CORBA compliant.

Most of the approaches for the development of distributed systems start with a new set of specifications or design. The approach that we defined has the singular characteristic that in this research we target the migration of existing legacy systems to distributed environments. Code availability offers some advantages. For example, some of the object-oriented metrics (e.g., DAC), programs-uses-file relationships, and dependence relations (e.g., control dependence) require information that is only available in late stages of the development process.

Other contributions from the methodology are:

- Incremental migration. An important feature is the incremental and evolutionary nature of the migration process. Since the subsystem decomposition produces

relatively independent subsystems (with high data cohesion and low coupling), it is possible to target, replace, reuse, or redevelop each specific subsystem one at a time. This feature is especially useful for large legacy systems where the migration of the whole system in one shot is infeasible. The subsystem decomposition produces subsets of the original legacy system that can be addressed independently.

- Program understanding and maintainability. The methodology produces set of relatively independent components from the legacy code. The study (program understanding) of the program now is simpler, given that we can target specific subsystems. In addition, the reverse engineering and architectural recovery phase produces documentation of the subject legacy system. Such documentation is often missing or not up-to-date in legacy systems.
- The use of data mining elicits meaningful information without previous knowledge of the domain. This feature is especially important when dealing with software systems that lack documentation. Data mining techniques can produce relevant information of the subject system without any previous knowledge of the system's domain. Indeed, the source code is the only source of information that the methodology requires to produce the subsystem decomposition.
- The combination of data mining association rules and clustering association coefficients allows to compare and contrast the results of the subsystem decomposition using association coefficients (e.g., Jaccard's coefficients) and coefficients derived from the support measure of the data mining phase. The

'support' as an association coefficient for the clustering is a novel approach for system decomposition.

- Software metrics such as DAC, RFC, and CBO that have been used for many purposes (e.g., evaluation of quality of design, quality code indicator) were shown to be useful for driving the decomposition of the system into subsystems with high-cohesion, low-coupling and low inter-site dependence features. These features are desirable features in a distributed system.
- Integer programming techniques used for the implementation of the allocation algorithms were shown suitable for producing an efficient allocation of components to different sites in the network. The allocation of components produced a system with low inter-site communication. The optimization problem is an NP problem, but with the use of solvers that implement bounce-and-bound algorithms for the solution of the integer programming problem, it becomes a tractable problem.

The methodology consists of several phases (e.g., reverse engineering, subsystem decomposition, object-adaptation, components wrapping, and allocation and use of middleware technologies). There is a significant amount of research in each of the fields corresponding to the different phases (e.g., reverse engineering, and subsystem decomposition), but most of the research only targets one phase. In this research, we integrate and extend the results from those fields and produce a comprehensive methodology for the migration of legacy systems to distributed environments. Then, our work is unique as compared to existing approaches that develop distributed systems starting from the analysis and specifications of the distributed system.

Table 6 shows an updated version of the table that we presented in Chapter 2 with an additional column that corresponds to our research. Our research targets the goals of migrating (object-oriented and non object-oriented) legacy systems to distributed object environments.

Table 6. Research contribution

	[Burd96] [Burd98]	[Snee96a] [Snee96b] [Snee98]	[DeLu97]	[Lakh99]	[Alka99]	[Bar99] [Farn099] [Dame99a] [Dame99b] [Sass99]	[Mont99] [Mont98a]	[Pura98a] [Pura98b]	[Bast96]	[Manc99] [Soud99]	Serrano [Serr99b] [Serr99d]
Goal											
Legacy Systems to OO	X		X								X
OO to Dist. Systems (DS)								X	X		X
Legacy Systems to Dist. Sys.		X								X	X
Restructuring non OO	X	X		X			X				
Restructuring OO					X	X					
Subsystem decomposition	X		X				X				X
Integration (of systems)		X								X	X
Optimal allocation in DS								X	X		X
Security DS										X	
Techniques											
Code dependences	X		X				X				X
Wrapping		X	X					X	X	X	X
Slicing			X	X							
OO Metrics			X		X	X					X
Clustering							X			X	X
Data Mining							X				X
Optimization								X	X		X
Integrated tool environment	X	X	X	X	X	X	X	X		X	X

6.4. Future Research

This research presented an approach for the migration of legacy systems to distributed environments. The approach consists of several sequential phases (steps) that go from the legacy system to the resulting distributed system. Some of those phases use techniques from different areas where there are different approaches (e.g., reverse engineering, subsystem decomposition, clustering analysis, wrapping, allocation of objects, and middleware technology). Each of the techniques applied in any specific phase is independent of the previous/following phase, giving the approach the flexibility to choose from several approaches. For example, different clustering

algorithms, data mining algorithms, or allocation/placement of objects algorithms can be used.

For the subsystem decomposition of legacy systems, we used ISA. ISA only considers programs-uses-files relationships. A program 'uses' a file if the program reads or writes the file. It does not take into account the frequency of uses or the number of parameters involved in the operation. In addition, ISA does not consider other relationships existent in the legacy system (e.g., program calls program). Adaptations of ISA to consider these features or adaptation of another decomposition technique are paths for future work.

Experimentation with different object-oriented metrics needs to be performed to see which metrics produce results that are more suitable. We experimented with few cases and discovered that the interaction matrix of the CBO metric produces better decomposition, but more experiments need to be done. In addition, experiments with interaction matrices consisting of combinations of several metrics can be done.

In our research, we did not target the issue of security in the resulting distributed system. Souder proposes an approach for securely integrating legacy systems in a distributed environment [Soud99]. Security in distributed systems is a very important issue, especially when sensitive data is distributed over the network, and is not in the control of one single computer or application. Extending our methodology with security features is an area for future research.

Our research targets the allocation of components to different sites (i.e., inter-site distribution). This research can be extended with efficient algorithms for intra-site distribution (e.g., parallel processors). In addition, the allocation technique that we

propose takes into account only static features of the network and the communication between subsystems. Estimating the network traffic through execution tracing would help produce more accurate allocation. Purao proposes the use of usage patterns and reference joins to estimate traffic volumes [Pura98b]. Future research would include the study and adaptation of better allocation algorithms.

Finally, the middleware technology and standards are constantly changing driven by industry and market forces. Our approach does not rely on any specific middleware technology (e.g., CORBA, DCOM, DNA, or RMI). Future experiments could include the implementation and comparison of performance of this approach with different middleware technologies.

REFERENCES

[Alex77] Alexander, C., A Pattern Language, Oxford University Press, 1977.

[Alka98] Alkadi, G., Carver, D., "The Application of Metrics to Object-Oriented Designs", Proceedings of the 1998 IEEE Aerospace Application Conference, 1998.

[Anqu99] Anquetil, N., Lethbridge, T., "Experiments With Clustering As a Software Remodularization Method", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1999.

[Apte97] Apte, C., "Data Mining: An Industrial Research Perspective", Computational Science and Engineering, IEEE Computer Society Press, 1997, pp. pp. 6-9.

[Amst98] Armstrong, M., Trudeau, C., "Evaluating Architectural Extractors", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1998.

[Bajc98] Bajcsy, P., Ahuja, N., "Location- and Density-Based Hierarchical Clustering Using Similarity Analysis", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 20, No. 9, 1998 Sep, pp. 1011-1015.

[Bar99] Bar, H., Bauer, M., Ciupke, O., Demeyer, S., Ducasse, S., Lanza, M., et.al., "The Famoos Object-Oriented Reengineering Handbook", 1999. Available at: http://www.iam.unibe.ch/~famoos/handbook/

[Bar98] Bar, H., Ciupke, O., "Exploiting Design Heuristics for Automatic Problem Detection", 12th European Conference on Object-Oriented Programming, Springer-Verlag, 1998. Available at: http://www.fzi.de/ECOOP98-W03/ecoop98ws.html

[Basi96] Basili, V., Briand, L., Melo, W., "A Validation of Object Oriented Design Metrics As Quality Indicators", IEEE Transactions on Software Engineering, Vol. 22, No. 10, 1996 Oct, pp. 751-761.

[Bast98] Bastarrica, M., Demurjian, S., Shvartsman, A., "Software Architectural Specification for Optimal Object Distribution", XVIII International Conference of the Chilean Computer Science Society, November 12-14, Antofagasta, Chile, 1998. Available at: http://dlib.computer.org/conferen/sccc/8616/pdf/86160025.pdf

[Bert90] Bertino, E., Negri, M., Pelagatti, G., Sbatella, L., "An Object-Oriented Data Model for Distributed Office Applications", Conference on Office Information Systems, ACM, 1990.

[Bran99] Brant, R., "Refactoring Browser", [Web Page]. Available at: http://st-www.cs.uiuc.edu/~brant/Refactory/RefactoringBrowser.html.

[Bria98b] Briand, L., Daly, J., Wust, J, "A Unified Framework for Cohesion Measurements in Object-Oriented Systems", Empirical Software Eng.: An Int'l J., Vol. 3, No. 1, 1998, pp. 65-117.

[Bria99a] Briand, L., Daly, J., Wust, J, "A Unified Framework for Coupling Measurements in Object-Oriented Systems", IEEE Transactions on Software Engineering, Vol. 25, No. 1, 1999, pp. 91-121.

[Burd98] Burd, E., Munro, M., "Assisting Human Understanding to Aid the Targeting of Necessary Reengineering Work", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1998, pp. 2-9.

[Burd96] Burd, E., Munro, M., Wezeman, C., "Extracting Reusable Modules From Legacy Code: Considering the Issues of Module Granularity", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1996, pp. 189-196.

[Ceri83] Ceri, S., Navathe, S., Wiederhold, G., "Distribution Design of Logical Database Schemas", IEEE Transactions on Software Engineering, Vol. 9, No. 4, 1983, pp. 487-504.

[Chae98] Chae, H., Kwon, Y., "A Cohesion Measure for Classes in Object Oriented System", 5th. International Symposium on Software Metrics, November 20-21, IEEE Computer Society Press, 1998. Available at: http://www.computer.org/conferen/proceed/metrics/9201/9201toc.htm

[Chid91] Chidamber, S., Kemerer, C., "Towards a Metric Suite for Object-Oriented Design", Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), Phoenix, AZ., ACM, 1991, pp. 197-211.

[Chid94] Chidamber, S., Kemerer, C., "A Metrics Suite for Object-Oriented Design", IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994, pp. 476-493.

[Chik90] Chikofsky, E., Cross, J., "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, Vol. 7, No. 1, 1990, pp. 13-17.

[CIMC99] Cimcom, "VisualWorks", [Web Page]. Available at: http://www.cincom.com/visualworks/.

[Cimi98] Cimitile, A., De Lucia, A., Di Lucca, G., "An Experiment in Identifying Persistent Objects in Large Systems", International Conference on Software Maintenance, Nov 16-20, Bethesda, Maryland, IEEE Computer Society Press, 1998, pp. 122-130.

[Data99] Data Destilleries, "Data Mining Solutions", [Web Page]. Available at: http://www.ddi.nl/.

[DeLu97] De Lucia, A., Di Lucca, G., Fasolino, A., Guerra, P., Petruzzelli, S., "Migrating Legacy Systems Towards Object-Oriented Platforms", Int. Conf. in Software Maintenance, IEEE Computer Society Press, 1997, pp. 122-129.

[Deme99a] Demeyer, S., Ducasse, S., "Metrics, Do They Really Help ?", Proceedings LMO'99 (Languages Et Modèles à Objets), Paris, France, HERMES cience Publications, 1999, pp. 69-82. Available at: http://www.iam.unibe.ch/~famoos/publications.html

[Deme99b] Demeyer, S., Ducasse, S., Lanza, M., "A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1999. Available at: http://www.iam.unibe.ch/~famoos/publications.html

[Deme99c] Demeyer, S., Ducasse, S., Tichelar, S., "Why FAMIX and Not UML ?", UML'99 Conference Proceedings, Springer-Verlag, 1999. Available at: http://www.iam.unibe.ch/~famoos/FAMIX/whyFAMIX/whyFAMIX.html

[Dewa99] Dewar, R., Lloyd, A., Pooley, R., Stevens, P., "Identifying and Communicating Expertise in Systems Reengineering: a Patterns Approach", IEE-Proceedings Software, 1999, pp. 145-164. Available at: http://www.reengineering.ed.ac.uk/

[Duca99] Ducasse, S., Richner, T., Nebbe, R., "Type-Check Elimination: Two Object-Oriented Reengineering Patterns", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1999.

[EIA 99] Electronic Industries Alliance, "CDIF - Case Data Interchange Format", [Web Page]. Available at: http://www.eigroup.org/cdif/index.html.

[Espr95] ESPRIT, "European Union Information Technology Program", 1995. Available at: http://www.cordis.lu/esprit/src/about.htm

[Famo99] FAMOOS, "European Commission ESPRIT Project 21975", 1999. Available at: http://www.iam.unibe.ch/~famoos/

[Fayy96a] Fayyad, U., "Data Mining and Knowledge Discovery: Making Sence Out of Data", IEEE Expert, Vol. 11, No. 5, 1996, pp. 20-25.

[Fayy96b] Fayyad, U., Piatetsky-Shapiro, G., Smyth, P., "The KDD Process for Extracting Useful Knowledge From Volumes of Data", Communications of the ACM, Vol. 39, No. 11, 1996, pp. 27-34.

[Fent97] Fenton, N., Pfleeger, S. Software Metrics: A Rigorous and Practical Approach, PWS Publishing Company, 1997.

[Four99] Fourer, F., "Linear Programming Frequently Asked Questions", [Web Page]. Available at: http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html.

[Gamm94] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.

[GAMS99] GAMS, "General Algebraic Modeling System", [Web Page]. Available at: http://www.gams.com/.

[Gann99] Gannod, G., Cheng, B., "A Framework for Classifying Software Reverse Engineering and Design Recovery Techniques", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1999.

[Geis94] Geist, A., Beguelin A., Dongarra J., Jiang, W., Manchek, R., Sunderam V., PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994.

[Guer99] Guerraoui, R., Fayad, M., "OO Distributed Programming Is Not Distributed OO Programming", Communications of the ACM, Vol. 42, No. 4, 1999, pp. 1001-104.

[Guja98] Guha, S., Rastogi, R., Shim, K., "CURE: an Efficient Clustering Algorithm for Large Databases", Int. Conf. on Management of Data, ACM, 1998, pp. 73-84.

[Harr98a] Harrison, R., Counsel, R., Nithi, R., "Coupling Metrics for Object Oriented Design", 5th. International Symposium on Software Metrics, November 20-21, IEEE Computer Society Press, 1998. Available at: http://www.computer.org/conferen/proceed/metrics/9201/9201toc.htm

[Harr98c] Harrison, R., Counsell, S., Nithi, R., "Design Metrics in the Reengineering of Object-Oriented Systems", 12th European Conference on Object-Oriented Programming, Springer-Verlag, 1998. Available at: http://www.fzi.de/ECOOP98-W03/ecoop98ws.html

[Hend96] HendersonSellers, B. Object Oriented Metrics, Englewoods Cliffs, New Jersey, Prentice Hall, 1996.

[IBM00] IBM, "Business Intelligence - Intelligent Miner", [Web Page]. Available at: http://www-4.ibm.com/software/data/busn-intel/index.html.

[IBM 98] IBM, Intelligent Miner for Data, IBM, 1998.

[Ishi97] Ishikawa, Y., Eds. Scientific Computing in Object Oriented Parallel Environments - First Int. Conf., ISCOPE 97, Springer, 1997, (Lectures Notes in Computer Science, v. 1343).

[JLC 92] JLC/CRM, "Reengineering Definitions", Policy Workshop SB-1 Joint Logistic Commanders Computer Resources Management Group, Software Technology Support Center, 1992. Available at: http://www.stsc.hill.af.mil/reng/defin.asp

[Kava98] Kavanagh, F., "CORBA or DCOM ?", Distributed Computing, DC Corporation, 1998. Available at: http://www.distributedcomputing.com

[Lakh99] Lakhotia, A., Deprez, J., "Restructuring Functions With Low Cohesion", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1999.

[Li 93] Li, W., Henry, S., "Object-Oriented Metrics That Predict Maintainability", J. Systems and Software, Vol. 23, No. 2, 1993, pp. 111-122.

[Lind99] Lindo Systems, "Lingo: The linear, non linear, and integer programming solver", [Web Page]. Available at: http://www.lindo.com/.

[Luck97] Lucksch, P., Mair, U., Rathmayer, S., Weidmann, M., Unger, F., "SEMPA: Software Engineering for Parallel Scientific Computing", IEEE Concurrency, Vol. 5, No. 3, 1997, pp. 64-72.

[Manc99] Mancoridis, S., Mitchell, B., Chen, Y., Gansner, R, "Bunch, A Clustering Tool for the Recovery and Maintenance of Software Systems Structures", Int. Conf. in Software Maintenance, Aug. 30 - Sep. 3, Oxford, England, IEEE Computer Society Press, 1999, pp. 50-59.

[Lanz99] Michele L., "Code Crawler", [Web Page]. Available at: http://www.iam.unibe.ch/~lanza/codecrawler/codecrawler.html.

[Micr98] Microsoft Corporation, "Distributed Component Object Model Protocol", [Web Page]. Available at: http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt.

[Mont99] Montes De Oca, C., Design Recovery and Data Mining: A Methodology That Identifies Data Cohesive Subsystems Base on Mining Associations Rules, Doctoral Dissertation, Louisiana State University, Dept. Computer Sciences, USA, 1999.

[Mont98a] Montes De Oca, C., Carver, D., "Identification of Data Cohesive Subsystems Using Data Mining Techniques", Int. Conf. in Software Maintenance, Nov 16-20, Bethesda, Maryland, IEEE Computer Society Press, 1998, pp. 16-23.

[Mozi00] Mozilla Organization - Netscape Corporation, "The Mozilla Web Browser", [Web Page]. Available at: http://www.mozilla.org/.

[OMG 99a] Object Management Group, "The Common Object Request Broker Architecture", [Web Page]. Available at: http://www.omg.org/corba.

[OTI99] Object Technology International Inc., "ENVY®/Developer", [Web Page]. Available at: http://www.oti.com/briefs/ed/edbrief5i.htm.

[Orac99] Oracle, "Oracle Darwin: Data Mining", [Web Page]. Available at: http://www.oracle.com/datawarehouse/products/datamining/.

[Ouya96a] Ouyang, Y., Carver, L., "Enhancing Design Reusability by Clustering Specifications", Symposium on Applied Computing, Feb. 18-20, Philadelphia, ACM, 1996.

[Ouya96b] Ouyang, Y., Carver, L., "A Model to Facilitate the Reuse of Specifications", Integrated Design and Process Technology, 1996.

[Pura98b] Purao, S., Jain, H., Nazareth, D., "Effective Distribution of Object Oriented Applications", Communications of the ACM, Vol. 41, No. 8, 1998, pp. 100-108.

[Renn98] RENAISSANCE, "European Commission ESPRIT Project 22010 ", 1998. Available at: http://www.comp.lancs.ac.uk/projects/RenaissanceWeb/index.html

[Ruga99] Rugaber, S., "A Tool Suite for Evolving Legacy Software", Int. Conf. in Software Maintenance, Aug. 30 - Sep. 3, Oxford, England, IEEE Computer Society Press, 1999, pp. 33-39.

[Sass99] Sassen, A., Marinescu, R., "Metrics-Based Problem Detection in Object-Oriented Legacy Systems Using Audit-Reengineer", 13th European Conference on Object-Oriented Programming, ECOOP, 1999. Available at: http://www.fzi.de/Ecoop99-WS-Reengineering/

[Sema98] SEMA Group, Concerto2/Audit, 1998.

[Serr99b] Serrano, M., Carver, D., "Migration of Object-Oriented Systems to Distributed Environments", International Conference on Parallel and Distributed Processing Techniques and Applications, Jun. 28 - Jul. 1, Las Vegas, NV., CSREA Press, 1999, pp. 403-409.

[Serr00] Serrano, M., Carver, D, Montes De Oca, C., "Mapping Object-Oriented Systems to Distributed Systems Using Data Mining Techniques", To Appear in The Thirteenth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, June 19 - 22, New Orleans, Springer Verlag, 2000.

[Simu96] Simoudis, E., "Reality Check for Data Mining", IEEE Expert, Vol. 11, No. 5, 1996, pp. 25-33.

[Snee96a] Sneed, H., "Encapsulating Legacy Software for Use in Client-Server Systems", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1996, pp. 104-119.

[Snee98] Sneed, H., Majnar, R., "A Case Study in Software Wrapping", Int. Conf. in Software Maintenance, Nov 16-20, Bethesda, Maryland, IEEE Computer Society Press, 1998, pp. 86-93.

[Soud99] Souder, T., Mancoridis, S., "A Tool for Securely Integrating Legacy Systems into a Distributed Environment", Working Conference in Reverse Engineering, IEEE Computer Society Press, 1999.

[Stev98] Stevens, P., Pooley, R., "Systems Reengineering Patterns", Proceedings ACM-SIGSOFT, 6th International Symposium on the Foundations of Software Engineering, 1998, pp. 17-23. Available at: http://www.reengineering.ed.ac.uk/

[Sun 99] SUN Microsystems, Inc., "JavaSoft Home Page", [Web Page]. Available at: http://www.javasoft.com.

[Take99] TakeFive Software, "The ABC's of SNiff+", [Web Page]. Available at: http://www.takefive.com/products/sniff+.html.

[TCSE97] TCSE, "Reengineering & Reverse Engineering Terminology", Technical Council on Software Engineering, IEEE Computer Society, 1997. Available at: http://www.tcse.org/revengr/taxonomy.html

[Pass99] University of Passau, "Graphlet", [Web Page]. Available at: http://infosun.fmi.uni-passau.de/Graphlet/.

[Vasu97] Vasudevan, P.,Shah, A., "World Wide Web - Beyond the Basics", [Web Page]. 1997. Available at: http://ei.cs.vt.edu/~wwwbtb/hardcopy/book/chap10/idl.html.

[Wals98] Walsh, A., Fronckowiak, J., Java Bible, IDG Books, 1998.

[Welc96b] Welch, L., Lankala, M., Farr, W., Hammer, D., "Metrics for Quality and Concurrency in Object-Based Systems", Annals of Software Engineering, Vol. 1, No. 1, 1996, pp. 93-119.

[Zion74] Zionts, S., Linear and Integer Programming, Prentice-Hall, Inc., 1974.

VITA

Miguel Angel Serrano-Vargas was born on October 18, 1970, in El Oro, México. He graduated with special honors in Computer Engineering from the Universidad Autónoma Metropolitana (UAM) in México City in May 1992. In 1994 he was granted a Fulbright-Conacyt scholarship/loan to pursue doctoral studies at Louisiana State University in Baton Rouge, Louisiana, in the United States of America.

Miguel Serrano has worked in both research and industry. From 1989 to 1994 he was working for research projects in the Centro Neuro Psico Pedagogico (CNPP) in México City, research related to the study of brain mapping, sleep disorders and evoked potentials. From 1991 to 1994 he also worked for Euristica Sistemas and ATIS de México doing consulting for Procter and Gamble and Comecial Mexicana. From 1995 to 2000 he has been working as a Teaching Assistant in the Computer Sciences Department in Louisiana State University. Miguel Serrano received the degree of Master of Science in Information Systems and Decision Sciences at Louisiana State University in August 1998. He also received a master's degree in Systems Science at Louisiana State University in December 1999. He expects to receive his degree of Doctor of Philosophy in Computer Science in May, 2000.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Miguel A. Serrano

Major Field: Computer Science

Title of Dissertation: Rengineering of Legacy Systems to Distributed Environments

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

March 24, 2000